OPSYS ____ SERIAL 12441

OMEGASOFT 6809 PASCAL
VERSION 2 LANGUAGE HANDBOOK

ORDER NO. MPCS2

RELEASED FOR SOFTWARE VERSION 2.2

CERTIFIED SOFTWARE CORPORATION
616 CAMINO CABALLO, NIPOMO CA 93444
TEL: (805) 349-0202 TELEX: 467013

Fifth printing, July 1983

CONTENTS

PREFACE

MANUAL OBJECTIVES

This manual describes the Pascal Language as implemented by OmegaSoft for the 6809 CPU. This manual is a reference document and is not meant as a tutorial on Pascal. For those not familiar with the Pascal Language the following book is recommended by OmegaSoft.

Programming in Pascal
Revised Edition
Peter Grogono
Published by Addison-Wesley
Number ISBN 0-201-02775-5

Readers should have a basic knowledge of the operating system to be used with the Pascal Compiler. Operating system dependant features are presented in the rear of this manual.

STRUCTURE OF THIS MANUAL

* Chapter 1 describes the basic structure of a Pascal program and its elements.

* Chapter 2 describes in detail the components of the various declaration sections.

* Chapter 3 describes the valid forms of an expression.

* Chapter 4 describes the statement types allowed in Pascal.

* Chapter 5 describes the format of user defined procedures and functions.

* Chapter 6 describes the predeclared functions supported.

* Chapter 7 describes the predeclared procedures supported.

* Chapter 8 describes Modular compilation and its advantages.

* Chapter 9 describes how you can interface assembly language data and programs with Pascal.

* Chapter 10 describes Pascal device drivers.

* Chapter 11 is an index for the preceding 10 chapters.

* Chapter 12 is an appendix that includes compilation and runtime errors, the runtime environment, the runtime support routines used and their location, and the ISO Validation Report. This report can be used as an aid for those of you who must write portable Pascal programs or wish to transport programs from another source to run on OmegaSoft Pascal.

* Chapter 13 includes a number of program examples developed at OmegaSoft or by its customers (reprinted here with their permission).

* Chapter 14 describes the working of the Symbolic Debugger included with the Pascal system.

* Chapter 15 describes any other utilities that are included. These utilities are operating system dependant.

* Chapter 16 outlines the installation of the Pascal system on your operating system. Before trying to use this software this section should be read. This chapter also provides an example program (source on disk) and a session where this program is tested using the debugger and then assembled and linked into an operating system utility.

* Chapter 17 details the operation of the standard I/O devices on your operating system.

INTRODUCTION

OmegaSoft Pascal is an extended implementation of the ISO standard Pascal language. This language is designed for industrial control and related real-time applications and features the following extensions :

* Additional data types including longinteger, longhex, hex, string, and user-defined device.

* Logical operators for 1, 2, and 4 byte types.

* Shift operator for 1, 2, and 4 byte types.

* Loop exit and program halt.

* Hex and binary constants.

* Easy interface to assembly language procedures, functions, and variables.

* Else clause and subrange constants for case statement.

* ADDR and SIZEOF functions.

* Arithmetic operators for one byte values.

* Include files.

* Modular compilation.

Features in the ISO standard not supported in this compiler are packed variables (and associated procedures) and procedures and functions as formal parameters to procedures and functions.

SYNTAX NOTATION USED IN THIS MANUAL

Modified Backus-Naur form is used as the primary syntax in this manual. Note that terminal symbols (reserved words, special symbols) are in boldface print.

Meta-Symbol	Meaning
=	shall be defined to be
|	alternatively
[x]	zero or one instance of x
{x}	zero or more repetitions of x
(x|y|..|z)	grouping: any one of x,y,..,z
xyz	the terminal symbol **xyz**
anything-else	a non-terminal symbol

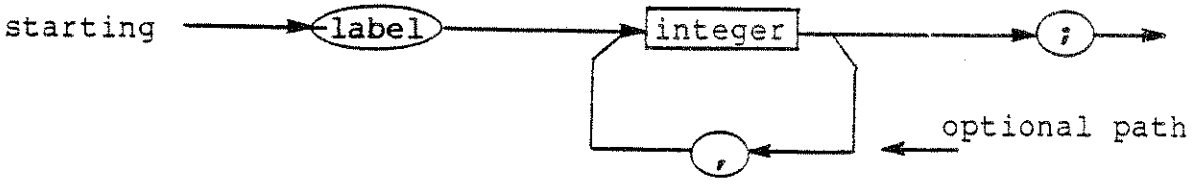
For more complex constructions pictorial syntax diagrams are also provided.

Ovals are used to represent reserved words, predefined identifiers, and symbols. These appear in the program as shown and are not expanded out into further syntax. The items within the ovals are in boldface print to correspond with the Backus-Naur form outlined above.

A rectangle contains syntax elements that are defined elsewhere in their own diagram. These can be either user defined identifiers or parts of the syntax that can be expanded out into its basic components.

Lines and arrows indicate authorized paths and are used to show the acceptable sequence of elements in the syntax diagram.

For example, the syntax for the label declaration is drawn as follows:



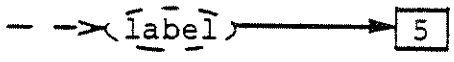
The correctness of the declaration

```
label 5,100 ;
```

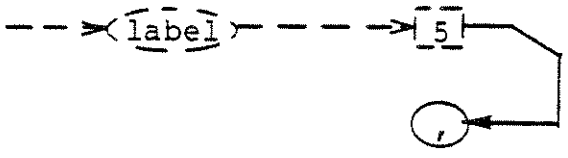
can be verified by tracing through the syntax diagram. The following diagram is a step-by-step illustration of how the label declaration is constructed (or verified) by following the syntax rules.

label

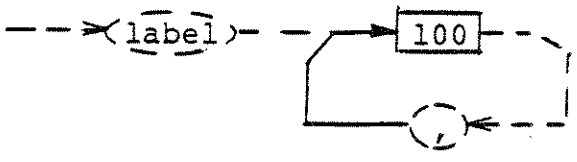
label 5



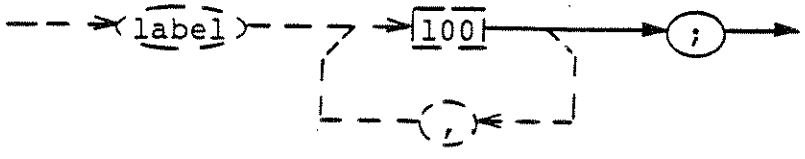
label 5,



label 5,100



label 5,100 ;



PASCAL PROGRAM STRUCTURE

PROGRAM HEADING

The PROGRAM line must be the first line of a program. An identifier follows the word PROGRAM and this identifier is converted to upper case, truncated to 6 characters, and used as the starting address label for the program. This identifier has no further significance in the program.

Following this identifier may be a list of standard I/O devices and device variable names. If any of the standard I/O devices are to be used in the program, then you must list them here.

If any device variable names are listed, then their relative position corresponds to a position within the command line. Using the program heading and variable declarations that follow :

```
program Test (data, report, error_file) ;
  var
    error_file, data, report : text ;
```

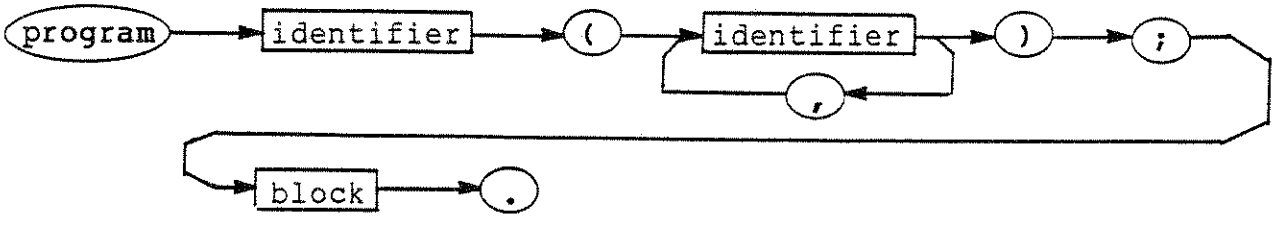
then if the program is called with the command line :

```
test jones jreport jerror
```

then the default filename for data will be jones, for report will be jreport, and for error_file will jerror. For additional information on options available for the standard I/O devices see the section on compilation options.

Following the optional list of identifiers is a semicolon, the outer block, and the program is terminated by a period.

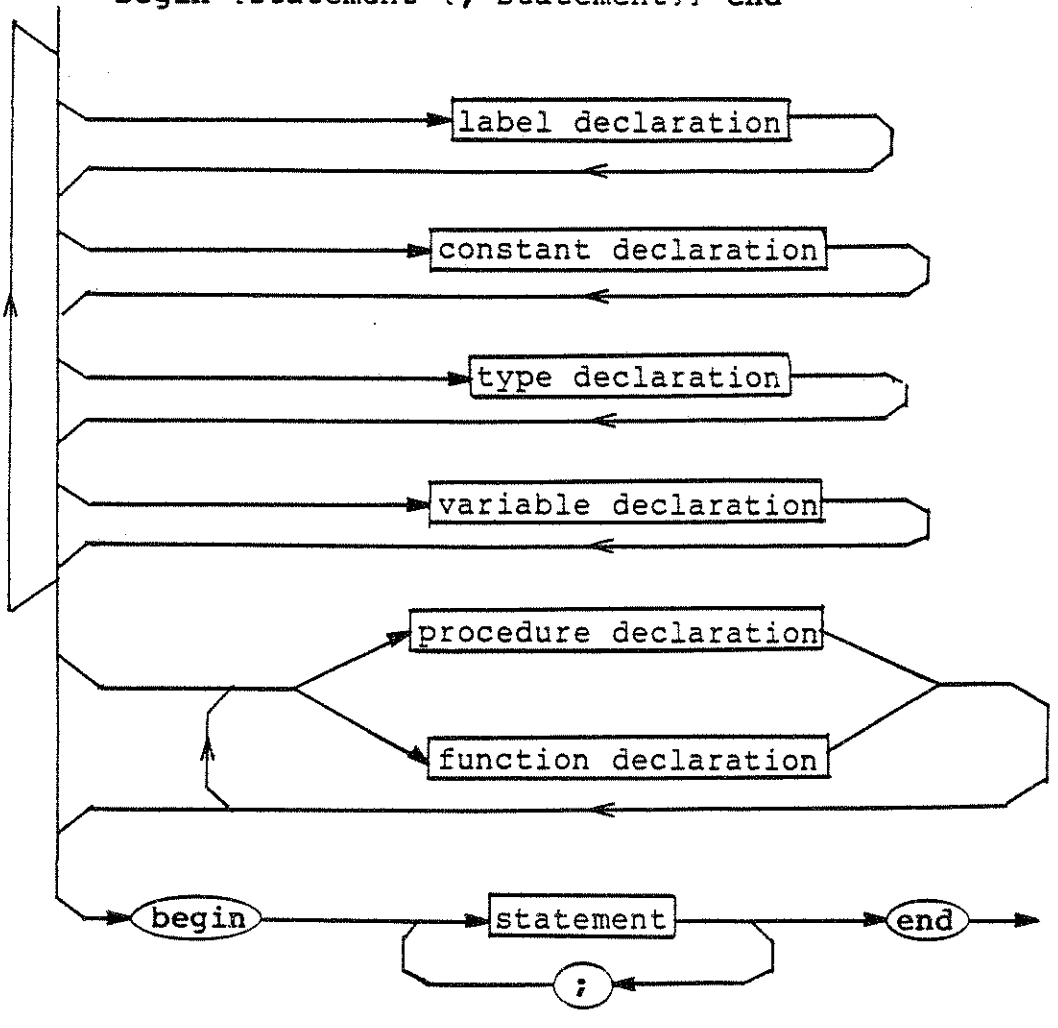
program heading = **Program** identifier [(identifier
{, **identifier**})] ; block .



BLOCK

The block contains the declaration part and execution part. All data items referenced in the execution part must be defined in the declaration part of the same or an enclosing block.

```
block = {(label-declaration | constant-declaration |
         type-declaration | variable-declaration)}
         {(procedure-declaration | function-declaration)}
         begin [statement {; statement}] end
```



LEXICAL LEVELS

Each block is at a specific lexical level. This lexical level is assigned by the compiler and is used to determine the code required at runtime to access variables. The first lexical level defined is the outer (program) block and is level one. Procedures and functions defined immediately within the program block are at lexical level two. Procedures and functions defined within lexical level two are at lexical level three, etc.

A procedure defined at lexical level "N" may call a procedure defined within its block at lexical level "N + 1". A procedure defined at lexical level "N" may also call any procedure at a lower lexical level in an enclosing block. This is shown in the following example.

```
Program A;    { lex level 1 }

  Procedure B; { lex level 2 }

    Procedure C; { lex level 3 }
      Begin
        { procedures B and C may be called }
      End;

    Begin { B }
      { procedures B and C may be called }
    End;

  Procedure D; { lex level 2 }

    Procedure E; { lex level 3 }

      Procedure F; { lex level 4 }
        Begin
          { procedures B, D, E, and F may be called }
        End;

      Begin { E }
        { procedures B, D, E, and F may be called }
      End;

    Begin { D }
      { procedures B, D, and E may be called }
    End;

  Begin { A }
    { procedures B and D may be called }
  End.
```

Constants, types, variables, and parameters carry the lexical level of the procedure (or program) they are defined in. When accessing a variable the difference between the current lexical level and the lexical level that the variable is defined at affects the amount of code (and time) required. The smallest amount of code is required for accessing local variables (those that are defined in the same block as the access). Global variables (defined in the program block, but accessed from a procedure) require more code to access. Variables defined in a procedure block and then accessed in an inner block require the most code to access.

CHARACTER SET

OmegaSoft Pascal uses the American Standard Code for Information Interchange (ASCII) character set. Of the ASCII character set the following characters are used :

* The upper and lowercase letters A through Z and a through z

* The numbers 0 through 9

* The special characters space ! # $ % ' ( ) * + , - . / : ; < = > @ [ ] ^ _ { }

* The carriage return (hex 0D) for terminating lines of input

Upper and lower case characters are equivalent except in character and string constants.

IDENTIFIERS

Identifiers are used to name programs, constants, types, variables, procedures, and functions. Some identifiers are predeclared by the compiler but can be redefined by the user. Other identifiers are predeclared and cannot be redefined by the user, these are called reserved words. An identifier is a sequence of letters, digits, and underscores (_) with the following restrictions :

* The identifier must begin with a letter

* The identifier must not contain any blanks

* The identifier must not cross a line boundary

OmegaSoft Pascal does not place a restriction on the length of an identifier and all characters are significant. The following are examples of valid and invalid identifiers :

VALID	INVALID	
test1	_test1	{starts with _}
test200	200test	{starts with a digit}
time_of_day	time&day	{contains an ampersand}

PREDECLARED IDENTIFIERS

OmegaSoft Pascal predeclares the following identifiers :

abs	addr	arccos	arcsin
arctan	auxout	boolean	break
byte	char	chr	cline
close	concat	conversion	cos
create	del	deverr	device
dispose	e	entry	enum
eof	eoln	exit	exp
external	false	floor	forward
get	halt	hex	index
input	integer	interactive	keyboard
length	ln	log	longhex
longinteger	mark	maxint	maxlint
memavail	minint	minlint	module
new	odd	open	ord
output	page	pi	pred
put	random	range	read
readln	real	release	reset
rewrite	round	seek	sin
sizeof	sqr	sqrt	string
substr	succ	tan	text
true	trunc	update	upshift
write	writeln		

These predeclared identifiers may be redefined to denote some other item. Doing so will mean that you will not be able to use the identifier for its intended purpose and therefore it is recommended that you not redefine standard identifiers.

The standard I/O device names : input, output, auxout, and keyboard are considered global if they are declared in the program.

RESERVED WORDS

Reserved words are predeclared identifiers that may not be redefined at any time and include :

and	array	begin	case
const	div	do	downto
else	end	eor	file
for	function	goto	if
in	label	mod	nil
not	of	or	packed
procedure	program	record	repeat
set	then	to	type
var	until	while	with

SPECIAL SYMBOLS

OmegaSoft Pascal includes the following special symbols :

Symbol	Usage
space	delimiter
!	inline assembly code marker
#	character constant marker
$	hex constant marker
%	binary constant marker
'	character and string constant delimiter
(	start of parameter list or expression
)	end of parameter list or expression
*	arithmetic multiply and set intersection
+	arithmetic addition and set union
,	list separator
-	arithmetic subtraction or set difference
.	end of program or decimal point
/	division
:	part of declaration syntax and case statement
;	general syntax separator
<	less than
<<	shift left
=	equal to
>	greater than
>>	shift right
<=	less than or equal to
>=	greater than or equal to
<>	not equal to
[	start of array index
]	end of array index
^	pointer to or pointer dereference
_	used in long identifiers for clarity
{	start of comment
}	end of comment
(*	alternate symbol for {
*)	alternate symbol for }
(.	alternate symbol for [
.)	alternate symbol for]
@	alternate symbol for ^

DELIMITERS

The semicolon (;) is used to separate one Pascal statement from another. More than one statement may appear on a line, but the statements must be separated by semicolons. Correct usage of semicolons tends to be the most common problem that new Pascal programmers have. It is unfortunate that Wirth made the semicolon a statement separator rather than a statement terminator. This methodology also applies to the use of the semicolon as a separator in declaration sections.

One way many Pascal programmers get away with poor use of semicolons is that a semicolon may be used immediately before the "end" in a compound statement or case statement. A semicolon in this position results in a "null" statement and is generally harmless.

The period (.) delimits the end of a Pascal program.

Spaces and carriage-returns are separators and cannot appear in an identifier, any number, or special symbol. At least one separator must be used between consecutive identifiers, reserved words, and numbers. This allows a great deal of freedom in the formatting, for an example see the next section on program documentation.

Begin and end are reserved words that act as delimiters. Begin indicates the start of a compound statement and is not followed by a semicolon. End terminates a record definition, a compound statement, or a case statement.

PROGRAM DOCUMENTATION

Probably the most significant advantage Pascal has over other languages such as assembly, "C", and Forth is its inherent self-documentation. This feature results in more source code for a given amount of object code than the other languages, but pays off in reduced maintenance costs in the long run.

Towards that end, it is recommended that your company define guidelines for program documentation and structure. The following are recommendations only and in no way are enforced by the compiler.

It is generally good practice to put a comment section before the main program and every procedure describing its purpose, parameters, and any global variables affected. It is usually not necessary to insert comments in with the code as is required by other languages.

The syntax for comments is :

```
comment = { characters } | (* characters *)
```

Comments are syntatically equal to separators. Replacing them by a blank does not alter the meaning of the program unless the comment is also a compiler control toggle (see compilation options section).

Since Pascal is a free format language, a great deal of freedom is provided for layout. All of the following represent the same program and will be compiled the same :

```
program
test
(output)
;
begin
writeln (
'test')
end.
```

```
program test (output) ; begin writeln ('test') end.
```

```
program test (output) ;
  begin
    writeln ('test')
  end.
```

As an aid to determining the structure of a program the compiler provides an "indent" count along with the line number on the listing. This indent count represents what level the compiler is in based on the first identifier in the line. This in no way dictates what style of layout you must use in your programs.

The recommended indenting is based on the guidelines presented in 'Programming in PASCAL' written by Peter Grogono and published by Addison-Wesley. The indent count starts at 0 representing the left margin and increments one per indenting level. After the count reaches 9 its next increment is A, and the count proceeds through the alphabet. If the count gets very far into the alphabet it is usually a signal that you should break something out into a procedure or function.

Const, type, var, procedure, and function headings should be indented. The bodies for the const, type, var, procedure, and function sections should be indented from their headings.

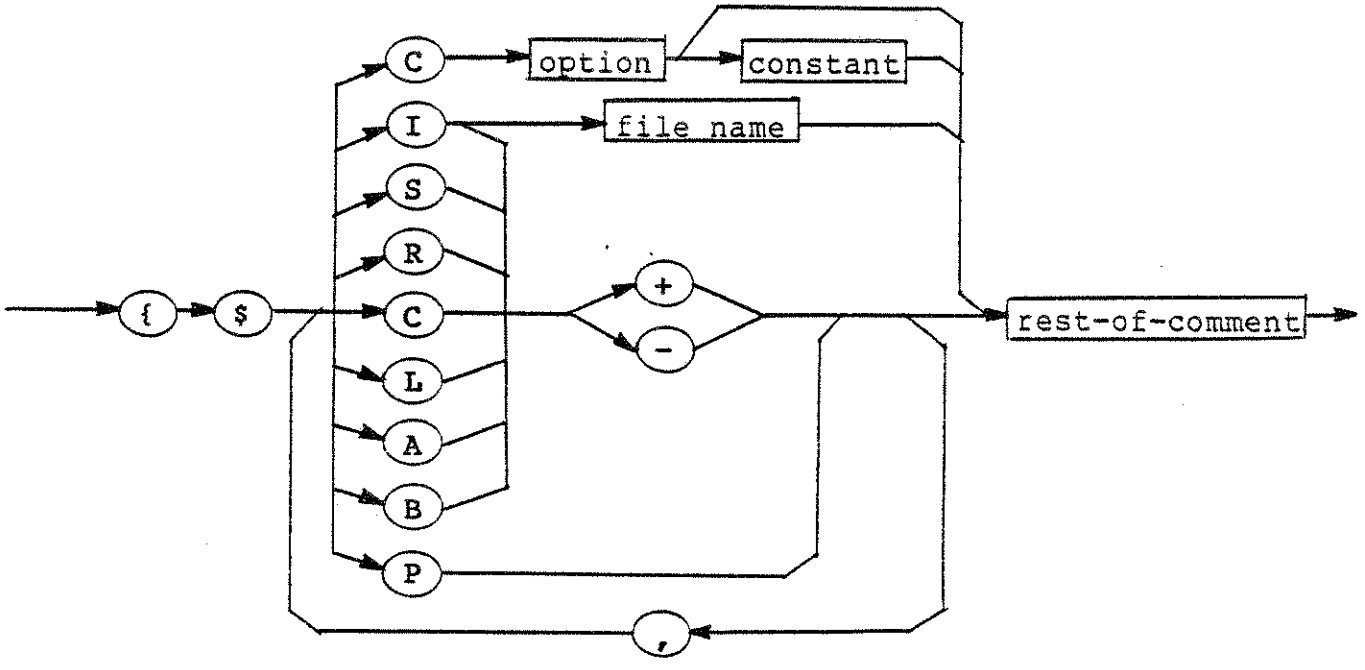
The following layouts are recommended for statements :

```
Begin
  Statement_1 ;
  Statement_2 ;
  ...
  Statement_N
End

If expression
  then
    statement

If expression
  then
    statement
  else
    statement

While expression do
  statement

For X := expression_1 to expression_2 do
  statement

Repeat
  Statement_1 ;
  Statement_2 ;
  ...
  Statement_N
Until expression

With variable do
  statement

Case expression of
  Constant1 :
    statement_1 ;
  Constant2 :
    statement_2 ;
  ...
  ConstantN :
    statement N
  Else
    statement
End { case }
```

COMPILATION OPTIONS AND INCLUDE FILES

In OmegaSoft Pascal there is a special syntax used to control various aspects of the compiler. If the open of a comment (a {) is followed by a dollar sign, then the characters that follow are considered to be compiler control toggles.

```
control-toggles = { $ ((l|s|b|c|a|r|i) (+|-)) | p)
                      {,((l|s|b|c|a|r|i) (+|-)) | p)} |
                      c (ne|eq|lt|le|gt|ge) constant | cend |
                      i<file name> rest-of-comment
```



Some of the options affect the action of the compiler during the compilation, other options generate code that will modify operation only at runtime and only after the code containing the options is executed.

The first set are the options that affect the operation of the compiler :

If the S option is followed by a + then subrange checks are enabled, if the S option is followed by a - then subrange checks are disabled. If subrange checks are on when doing assignments to subrange variables extra code is emitted to perform this check. This option also affects the type of code generated for array and string indexing and whether or not code is generated to check for truncation and overflow errors on arithmetic done in inline code rather than a subroutine at runtime. Note that it does no good to have the S option on without the R option on.

If the I option is followed by a file name, this file is opened and source text is read and compiled until end of file is hit, at which point the original file is used again. This syntax is referred to as an "Include file" and they may not be nested. If compiling an include file the compiler will place an "I" to the left of the line number on the listing.

If the P option is specified and the listing page size is non-zero then the next line of source will appear on a new page.

If the L option is followed by a + and the L command line option is on then listing will be enabled. If the L option is followed by a - and L command line option is on listing will be disabled. Note that these can be nested, If two L- 's are encountered then two L+ 's will be required to enable the listing. Default at start of compilation is L+.

If the B option is followed by a + then any text devices that are declared after that point will have their "break bit" set during initialization. If the B option is followed by a - then any text devices that are declared after that point will have their "break bit" cleared during initialization. Default at start of compilation is B-. Refer to the section of Chapter 2 dealing with Devices and files for an explanation of the "break bit".

The A option is not useful for OS-9 systems and this section can be skipped if you will be running your target program under OS-9. The A option is only effective during the program parameter list declaration. If enabled (A+) then any standard I/O devices defined there after will be setup for automatic command redirection. If disabled (A-) then any standard I/O devices defined there after will be setup for fixed attachment (they will use their normal devices). The default at the start of compilation is A-. If the A option is enabled then the strings that follow the special symbols in the command line will be the device or file name that will be used for the standard I/O device.

Input uses string following <
Output uses string following >
Auxout uses string following >>

If the string is P or p then it will connect to the system printer, if the string is TERM or term then it will connect to the system terminal, any other string will connect to a disk file with the string as its directory name.

If the C options is followed by "ne" "eq" "lt" "le" "gt" "ge" and a constant, this is the start of conditional compilation. The constant must either be a numeric one or two byte constant, or an identifier declared in the constant declaration section. This constant value is compared (signed) against zero, if it meets the condition specified then compilation will continue, else compilation will be turned off (source not parsed) until a "cend" compilation option is encountered. Conditional compilation may not be nested.

The other group of options generate runtime code that affects the error mask in the stack frame during execution. These options must be located in a section of the program that will be executed. Although there is sometimes executed code outside of a blocks begin .. end pair, it is not recommended that you put these options anywhere other than in a block's begin .. end pair.

When a procedure is entered its error mask is copied from its lexical parent, not the caller (unless the same).

If the I option is followed by a + then runtime I/O checks are enabled, if the I option is followed by a - then runtime I/O checks are disabled. Default at start of compilation is I-.

If the R option is followed by a + then runtime range checks are enabled, if the R option is followed by a - then runtime range checks are disabled. Default at start of compilation is R-.

If the C option is followed by a + then runtime conversion checks are enabled, if the C option is followed by a - then runtime conversion checks are disabled. Note that this only refers to halting the execution on a conversion error, the conversion function will always return the latest status, regardless of the condition of this toggle. Default at start of compilation is C-.

A brief example will more clearly demonstrate the effect of the I, R, and C options. Only the R option will be used since the other two behave exactly the same.

```
Program A ;
  Procedure B ;
    Procedure C ;
      Begin
        { at this point options are the same as Procedure B
          at the time when Procedure C is called. }
        {$R+ this affects only Procedure C }
        B { its options will reflect Program A's
            because of the lexical levels }
      End ;
    Begin { B }
      { at this point options are the same as Program A
        at the time when Procedure B is called. }
      {$R- this does not affect Program A }
      C  { at entry Procedure C's R flag will be off }
    End ; { B }
  Begin { A }
    { at this point all options are off }
    {$R+ turn on range checks }
    B { at entry Procedure B's R flag will be on until
        set explicitly off }
  End . { A }
```

DECLARATION SECTION

All constant, type, variable, and procedure names must be declared before being used. This chapter describes each of the declaration sections available in OmegaSoft Pascal.

Before covering the declaration sections it is appropriate to present the number types available in OmegaSoft Pascal.

NUMBERS

OmegaSoft Pascal recognizes integer (signed fixed point), hex (unsigned fixed point) and real (signed floating point) numbers. Hex numbers may be represented in hexidecimal (0-9 and A-F) form or binary (1's and 0's) form. In addition, integer and hex numbers may be used to form byte (unsigned fixed point) numbers.

INTEGERS

Normal integers are positive and negative whole numbers ranging from -32768 to 32767. A minus sign (-) precedes a negative integer, a plus sign (+) may precede a positive integer, but has no effect.

Longintegers are positive and negative whole numbers ranging from -2147483648 through 2147483647. Minus and plus signs are allowed in the same manner as normal integers. A longinteger is an integer that is out of the integer range, or an integer followed by an "L" with no intervening space.

HEX

Hex numbers are formed by using the decimal digits 0 through 9 and the letters A through F preceded by a dollar sign ($). The range for hex numbers is $0 through $FFFF.

Hex numbers may also be represented by using the decimal digits 0 and 1 preceded by a percent sign (%).

Longhex numbers have a range of $0 through $FFFFFFFF. A longhex number is a hex number that is out of the hex range, or a hex number followed by an "L" with no intervening space.

BYTES

Byte numbers are formed by prefixing a pound sign (#) in front of an unsigned integer or hex number. Only the least significant byte of the integer or hex number will be used. Byte numbers and byte variables are identical to character numbers and character variables, byte and char are synonyms.

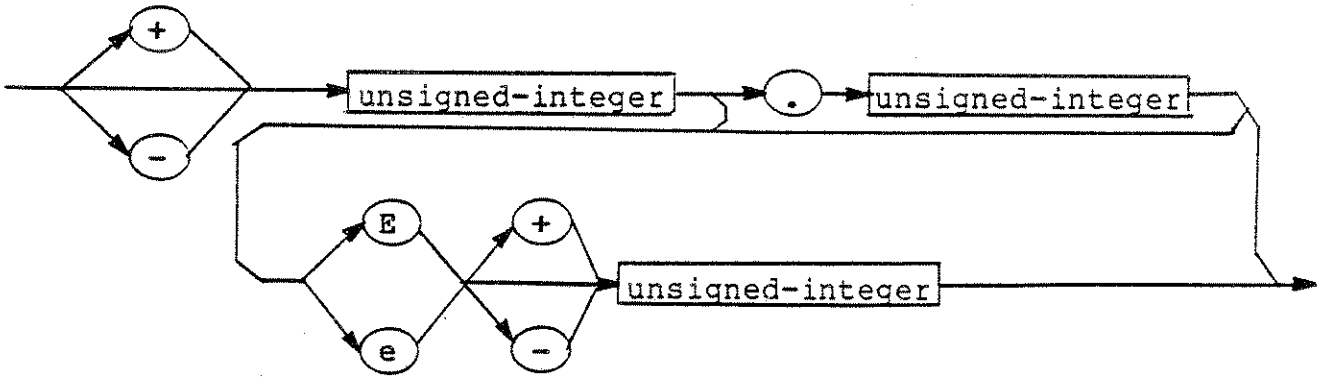
REALS

Real numbers have a range of approximately 2.7E-20 to 9.2E18 and a resolution of 7 digits. Note that due to the algorithms used the compiler and runtime routines will not convert the ascii representation of a real number into internal binary format outside of the range 5E-19 to 5E18.

The cumulative syntax for the above numbers follows :

```
digit-sequence = digit {digit}
unsigned-integer = digit-sequence
unsigned-real = unsigned-integer ( . unsigned-integer
                [(e | E) signed-integer] | (e | E) signed-integer)
hex-digit = (digit | A | B | C | D | E | F)
hex-number = ( $ hex-digit {hex-digit} | % (0 | 1) {(0 | 1)} )
byte-number = # (unsigned-integer | hex-number)
signed-integer = [(+ | -)] unsigned-integer
signed-real = [(+ | -)] unsigned-real
longinteger = signed-integer [L | l]
longhex = hex-number [L | l]
```

Pictorial diagram for signed-real :



LABEL DECLARATIONS

The label declaration is used to define to the compiler those unsigned-integers to be used as destinations for a GOTO statement. Note that this section does not define where the label will be, only that the designated values may be used as a label.

label-declaration = **label** unsigned-integer {, unsigned-integer} ;

CONSTANT DECLARATIONS

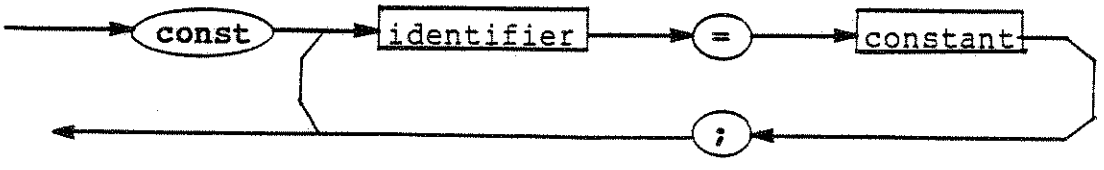
Constants are identifiers that are assigned an unchanging value. Constant identifiers are preferred over the use of numbers within a program. OmegaSoft Pascal has a very useful extension which allows a constant to be any constant-valued simple expression (see Chapter 3 for details of simple-expression syntax).

The type of the simple expression must be boolean, character, integer, longinteger, hex, real, string, or pointer.

The following types and operators and functions will remain constant if the operands are constants :

Not operator for boolean, character, integer, and hex. Unary negation for integers, longintegers, and real. Addition and subtraction for character, integer, and hex. Or operator for boolean, character, integer, and hex. Eor operator for boolean, character, integer, and hex. Multiply for character, integer, and hex. And operator for boolean, character, integer, and hex. Shift operator for character, integer, and hex. Ord, and chr functions. Boolean function for enum, char, integer, and hex parameters. Enum and Char functions for integer and hex. Integer and Hex functions for boolean, enum, and char. Longhex function for longinteger, Longinteger function for longhex parameter. Above general conversion functions when the parameter is the same type as the function.

constant-declaration = **const** identifier = constant
{; identifier = constant} ;
constant = constant-valued-simple-expression



PREDECLARED CONSTANTS

These are predeclared identifiers having specific constant values, they include :

false	- boolean with an ordinal value of 0
true	- boolean with an ordinal value of 1
maxint	- integer with a value of 32767
minint	- integer with a value of -32768
nil	- pointer or hex with a value of $0
maxlint	- longinteger with a value of 2147483647
minlint	- longinteger with a value of -2147483648
e	- real with a value of 2.718282
pi	- real with a value of 3.141593

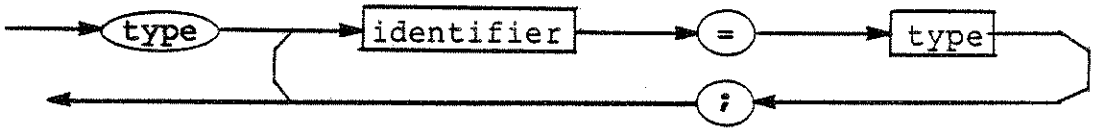
TYPE DECLARATIONS

The type-declaration part is used to assign a type to an identifier, or to create a new type out of existing types, either user defined, or predeclared.

In this chapter we will also describe how the various types are stored in memory as variables an how they appear on the data stack during expression evaluation or when passed as parameters.

The simplest of the types are the ordinal types. Any of the ordinal types can be converted into a unique integer. The ordinal types include : boolean, character, enumerated, integer, hex, and subranges.

type-declaration = **type** identifier = type {; indentifier = type};



BOOLEAN

Boolean types represent logical values, either false or true. Boolean is a predeclared enumerated type :

boolean = (false, true)

such that in relational expressions : false < true.

Boolean variables are stored as one byte. The most significant 7 bits of the byte must be zero. If the least significant bit is zero, then it is false, else it is true. When boolean values are used in expressions they are in the B accumulator. When boolean values are passed as parameters by value or returned from a function they occupy one byte on the stack.

boolean-type = **boolean**
boolean-constant = (**false** | **true** | boolean-constant-identifier)

CHARACTER AND BYTE

A type defined as char (for character) or byte are exactly the same, char and byte are synonyms. Character values most often represent ascii data, but may represent any other data that can fit within one byte. Character variables are very useful for interfacing to byte-wide I/O ports by using absolute addressing.

Character variables are stored as one byte. When character values are used in expressions they are in the B accumulator. When character values are passed as parameters by value or returned from a function they occupy one byte on the stack.

```
character-type = char | byte
character-constant = (' character ' |
                      # (unsigned-integer | hex-number) |
                      character-constant-identifier)
```

Note that in a character-constant if it is necessary to represent the single quote (') it must be written twice ('').

ENUMERATED

An enumerated type declaration specifies the permissible values for that type. Enumerated types are most often used as selectors for case records and statements, or as values for array indexes. They have the advantage over using integers or bytes in that each value can be made to represent what it will actually be doing. Enumerated type values such as (current, voltage, phase) would be much more meaningful than using the values 0, 1, 2.

Enumerated type variables are stored as one byte, the first value listed in the declaration is assigned to zero, the next to one, etc. When enumerated types are used in expression they are stored in the B accumulator. If an enumerated type is passed as a parameter by value or returned from a function they occupy one byte on the stack.

The identifier names given are not actually used at run-time, therefore they cannot be read or written. Enumerated type is equivalent to the user defined scalar type as described in the Jenson and Wirth report.

```
enumerated-type = ( identifier {, identifier} )
```

The identifiers listed become constants of that enumerated type.

```
enumerated-constant = identifier
```

INTEGER

Integer variables are stored in two bytes, the most significant byte being at the lower address. When integers are used in expressions they are in the D accumulator, with the most significant byte in the A accumulator. When integers are passed as parameters by value or returned from a function they occupy two bytes on the stack, the most significant byte being at the lower address.

```
integer-type = integer
integer-constant = signed-integer | integer-constant-identifier
```

HEX

Hex types are used to represent unsigned 16 bit values such as addresses. Hex variables are stored in two bytes, the most significant byte being at the lower address. When hex values are used in expressions they are in the D accumulator, with the most significant byte in the A accumulator. When hex values are passed as parameters by value or returned from a function they occupy two bytes on the stack, the most significant byte being at the lower address. Note that hex values are also compatible with any pointer variable.

hex-type = **hex**
hex-constant = hex-number | hex-constant-identifier

SUBRANGES

Subranges specify an ordinal type with a reduced range. Subranges are used to specify array indices, and in cases where you want to restrict the range of an ordinal type. When an assignment is made into a subrange variable (with the compiler S option enabled) extra code is generated to make sure that the value to be assigned does not fall outside of the subrange declaration. The following are examples of subranges of the ordinal types :

```
boolean :       false .. true
character :     'A' .. 'Z'      #0 .. #$1F
enumerated, with definition of (one,two,three) :    two .. three
integer : -32768 .. 32767     0 .. 10       -5 .. 5
hex : $0 .. $FF      $1000 .. $7FFF
```

subrange-type = constant .. constant

LONGINTEGERS

Longintegers are used for applications where an integer does not have enough range and reals are not desirable due to either speed or roundoff problems. For instance in business applications where it is desirable to carry large money amounts and still have accurate cents amounts, reals are not suitable. A longinteger in ths application has a maximum value of $21,474,836.47 .

Longinteger variables are stored as four bytes, with the most significant byte being at the lowest address. When longinteger values are used in expressions, passed as parameters by value, or returned from a function, they occupy four bytes on the stack with the most significant byte being at the lowest address.

longinteger-type = **longinteger**
longinteger-constant = signed-integer |
longinteger-constant-identifier

REALS

Real numbers are used for applications requiring a large range with limited resolution.

Real variables are stored as four bytes, with the most significant (exponent) byte being at the lowest address. When real values are used in expressions, passed as parameters by value, or returned from a function, they occupy four bytes on the stack with the most significant byte being at the lowest address.

Real value format : (same as AMD9511)

```
          exponent          mantissa

        M  E
        S  S  EEEEEE  MMMMMMMMMMMMMMMMMMMMMMM

 bit    31 30 29  24  23                     0
```

The mantissa is expressed as a 24-bit unsigned normalized fractional value. The exponent is expressed as a two's complement 7-bit value having a range of -64 to 63. The most significant bit is the sign of the mantissa (0 = positive, 1 = negative), for a total of 32 bits. The binary point is assumed to be to the left of the most significant mantissa bit (23). Bit 23 must be equal to 1 (normalized) except for zero (all bits = 0).

```
real-type = real
real-constant = signed-real | real-constant-identifier
```

STRUCTURED TYPES

The structured types include strings, arrays, records, sets, and devices and files. The reserved word "packed" may precede any of these type declarations, but has no effect in OmegaSoft Pascal.

STRINGS

String types are most often used to represent strings of ascii characters. A unique feature of OmegaSoft Pascal is that each character in a string may be any 8 bit value, there is no special value used as a terminator. This implies that strings can be used for any variable length data up to 126 bytes in length.

String types are declared by following the identifier "string" by an integer number in brackets. The integer number defines the string's "static length", or longest possible length. Optionally, this integer in brackets can be left off, and the static length will be assumed to 80.

A string consists of a 1 byte dynamic length "n" in the range of 0 .. 126 followed by "n" bytes of characters. String variables occupy the declared length of the string plus one byte. The current dynamic length is at the lowest address, followed by the actual data. If the current string is shorter than the declared length, then any bytes following the string are garbage.

When a string value is used in expressions or returned from a function they occupy the current dynamic length of the string plus one byte on the stack. Even if the string is declared as 100 bytes, the string 'ABC' will occupy only 4 bytes on the stack. The dynamic length will be at the lowest address on the stack.

When a string value is passed as a parameter by value then the string will occupy the declared length of the parameter declaration plus one byte on the stack. The dynamic length will be at the lowest address on the stack. As an example, for the procedure declaration :

```
procedure a (b : string [40]) ;
```

If the string 'ABC' were to be passed to this procedure it would occupy 41 bytes on the stack. The lowest addressed byte would have the dynamic length (3), the next three addresses would have A, B, and C, and the next 37 bytes would be garbage. This is done so that the called procedure can access the parameter at a fixed offset from the stack mark, regardless of the size of the parameter.

```
string-type = string [ [ unsigned-integer ] ]
string-constant = ' { character } ' | string-constant-identifier
```

Note that in a string-constant if it is necessary to represent the single quote (') it must be written twice ('').

ARRAYS

An array is a group of variables all of the same type. Each variable in the array is called an element of the array and they are accessed by using the array identifier name and an index (or subscript). The index of an array can be any ordinal type except full range integer or hex. The element of the array can be any type.

The array is defined by following the reserved word "array" by the index declaration in brackets, in most cases this a subrange. After the index declaration the reserved word "of" is used, followed by the element type declaration.

An array variable must not occupy larger than 32757 bytes of memory. This byte count is determined by multiplying the size of the element times the range of the index, for instance :

```
array [0..31] of string [15]
```

occupies 32 * 16 = 512 bytes.

The above array variable would occupy 512 bytes in memory, either as a variable, or on the stack if passed as a parameter by value. The above array could not be returned from a function, but if the size of an array is less than 127 bytes, it can be a function return value. Two arrays are compatible if their size is equal.

If the element of an array is another array, then a multidimensional array is created. In OmegaSoft Pascal there is no limit on the number of dimensions possible, as long as the maximum size limit for the outer array is honored. There is a shorthand method of specifying multidimensional arrays :

```
array [1..5] of array ['A'..'C'] of integer ; is equivalent to :

array [1..5, 'A'..'C'] of integer
```

Multidimensional arrays are stored with the last index changing most rapidly. In the above array, the elements would be stored in memory as :

1-A 1-B 1-C 2-A 2-B 2-C 3-A 3-B 3-C 4-A 4-B 4-C 5-A 5-B 5-C

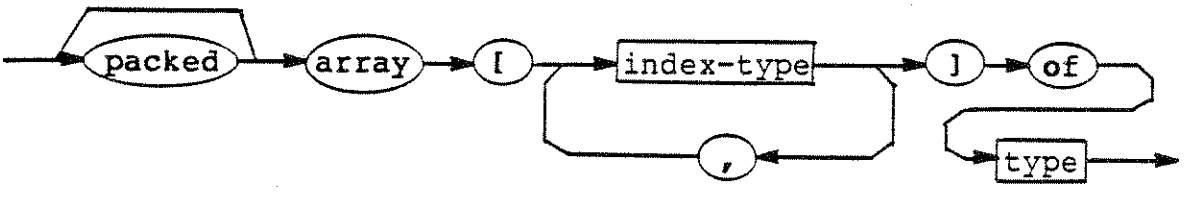
In ISO standard Pascal there is a concept called a character string variable defined as :

```
packed array [1..n] of char
```

This variable can be used as a sort of "string" in that it can accept string constants and be written out to a text file. The catch is that ALL of the characters are valid at all times, so this is not as good as the OmegaSoft Pascal string type. Note that if you must use this form that when assigning a string constant to this array that the length of the constant must match the declared length "n" or there will be garbage in the variable that will be considered valid.

OmegaSoft Pascal supports this form of string by allowing it to be written to a text file, and by making arrays of less than 127 bytes compatible with string constants and string variables.

```
index-type = ordinal-type
array-type = [packed] array [ index-type {, index-type} ]
             of type
```



RECORDS

Records are used to organize different data types into a new data type. Records have a fixed number of elements called fields. Fields consist of a fixed part (i.e. fixed size and definition) and a variant part (variable size and definition).

Each field in the fixed part of the record is allocated space immediately following the previous field. Each variant part starts allocation at the same location, therefore each variant uses the same space in a record. The tag field of the variant part is allocated space (if it is a variable definition) before the variants are allocated space. As an example of the allocation used, in the record :

```
alloc = record
          a, b : hex ;
          c : real ;
          d : string [4] ;
          case e : integer of
            0 : (f : real) ;
            1 : (g : string [4]) ;
            2 : (h : record
                       i,j : char
                     end)
        end ;
```

the fields use the following memory locations (relative to the start of the record) :

```
a : 0 - 1          b : 2 - 3          c : 4 - 7
d : 8 - 12         e : 13 - 14        f : 15 - 18
g : 15 - 19      h.i : 15           h.j : 16
```

The tag type must be an ordinal type and the tag constants must be the same type. The particular tag constants have no meaning other than signaling the start of a new variant. The variant part must be defined after the fixed part, but variant parts can nested. Variable identifiers must be unique within a record only. A field in a record is referenced by the record name followed by the field name.

The size of a record is the size of the fixed part plus the size of largest variant part, either as a variable, or on the stack if passed as a parameter by value. A record can be returned from a function if its size is less than 127 bytes. No record can exceed 32757 bytes.

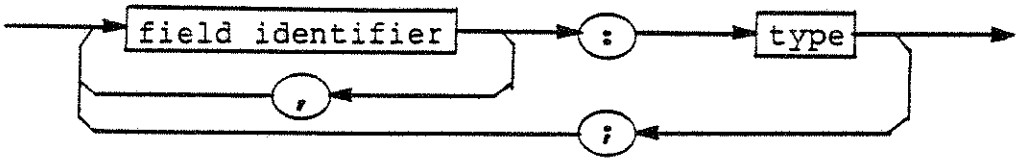
Two records are compatible if their sizes are equal.

```
record-type = [ packed ] record [field-list [;]] end
field-list = [ fixed-part ] [ ; variant-part ] | variant-part
fixed-part = record-section { ; record-section }
record-section = identifier {, identifier } : type
variant-part = case variant-selector of variant { ; variant }
variant-selector = [tag-field :] ordinal-type
tag-field = identifier
variant = constant {, constant} : ( [field-list [;]] )
```

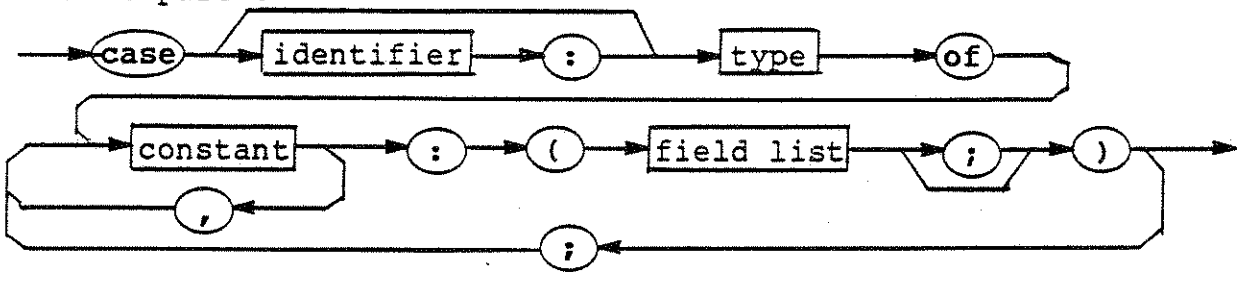
Record-type :

Field list :

Fixed part :



Variant part :



SETS

Sets are a collection of ordinal data items. Each set can have up to 1008 elements with ordinal values from 0 to 1007.

Sets are defined by following the reserved words "set" "of" with the base type of the set. The maximum ordinal value that the base type can have determines the amount of memory allocated for the set. The size is = maximum ordinal value div 8 + 2.

Sets are useful for collecting a series of attributes together, and to be able to insert and remove attributes without affecting the other attributes. For instance, in an assembler you could have an enumerated type :

```
stat = (declared,forward,encountered,absolute,relative) ;
```

and a set variable :

```
status : set of stat ;
```

You would then be able to set in the encountered attribute by using :

```
status := status + [encountered] ;
```

without affecting any of the other attributes, likewise you could remove the forward attribute by :

```
status := status - [forward] ;
```

One of the most common use of sets is in characters sets. For instance to check if a character is a letter, digit, or an "_" you could use a check like :

```
if upshift (ch) in ['A'..'Z','0'..'9','_'] then .....
```

which is much nicer than the multiple range checking alternative.

A set physically consists of a dynamic set length "n" (first byte) in the range of 0 .. 126 followed by "n" bytes. The lowest (ordinal value) element in the set is the least significant bit in the last byte of the set. The last byte contains elements 0 through 7, the second to the last byte contains elements 8 through 15 (element 8 is LSB), etc. If the current set is shorter than the amount allocated for the set, then any bytes following the "n" bytes are garbage.

When a set value is used in expressions or returned from a function they occupy the current dynamic length of the set plus one byte on the stack. Even if the set is declared as requiring 100 bytes, the set [4,12,19] will occupy only 4 bytes on the stack. The dynamic length will be at the lowest address on the stack.

Any set is compatible with any other set as long as an assignment is not made to a set variable that would require more memory than allocated to that variable.

When a set value is passed as a parameter by value then the set will occupy the declared length of the parameter declaration plus one byte on the stack. The dynamic length will be at the lowest address on the stack. As an example, for the procedure declaration :

```
procedure a (b : set of 0..250) ;
```

If the set [4,12,19] were to be passed to this procedure it would occupy 33 bytes on the stack. The lowest addressed byte would have the dynamic length (3), the next three addresses would have $08,$10,$10, and the next 29 bytes would be garbage. This is done so that the called procedure can access the parameter at a fixed offset from the stack mark, regardless of the size of the parameter.

```
set-type = set of ordinal-type
set-constant = see chapter 3 under set expressions
```

DEVICES AND FILES

A device is a sequence of data items, all with the same type, called components or elements of the device. The elements of a device can be any type, except for another device or any structure containing a device. Devices are the entity that represent I/O devices in a system.

A file is a special type of device normally associated with a random access disk, that have names that go with a specific group of data on the device. Procedures are included in Pascal to access these groups of data by their "file name".

A device consists of zero or more elements, only one of which can be accessed at any particular moment. The element is stored in a series of one or more locations in the device descriptor called the element buffer. This serves as a "window" through which the Pascal program can read or write to a device. The number of elements is not fixed and can vary up to the amount allowed for the particular device.

For every device declared there is automatically another variable which is called the buffer variable. This buffer variable points to the element buffer allowing the data to be transferred in and out of the device through the "window".

Devices are automatically initialized upon entering the block in which they are defined. For the standard devices (Input, Output, Keyboard, and Auxout) this means upon entering the main program block. For all but the standard devices, the device should be closed when no further accesses will be done.

All devices are assumed to be system level, they don't magically disappear when a block is exited. If you want a temporary file, delete it when you are done with it.

Text devices have the base type of character but in addition have line structure. The Procedures Readln, Writeln, and Page and the function Eoln are only valid for text devices. Any other procedures and functions that are available for non-text devices are also available for text devices. ASCII nulls are skipped when reading data from text devices by the device driver. Carriage returns are represented by a space in the element buffer and the eoln function returning true.

Interactive devices differ from other devices in that there need not be a valid element for the Eof and Eoln functions. The test is done based on the current flags. This was implemented so that in the situation as depicted below :

```
While not Eof do
  begin
    Write ('Enter value : ') ;
    Read (value) ;
    Writeln ('Log is : ', Log(value))
  end ;
```

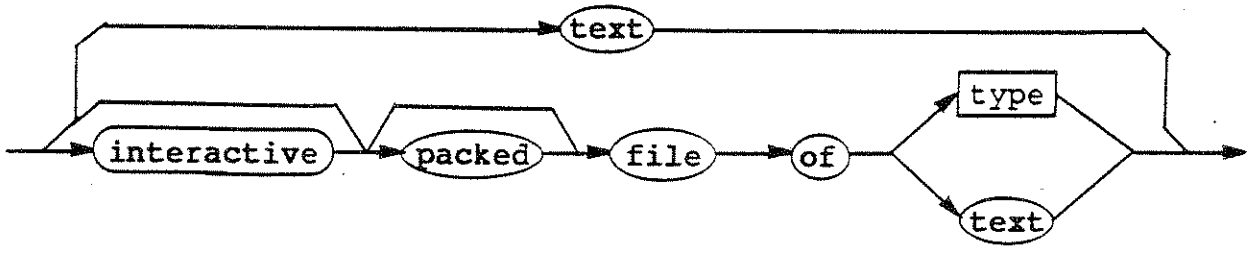
The first character of value need not be entered before you are told what you are supposed to enter by the Write statement as would be required in standard Pascal. Using this flag in the device descriptor we are able to use the same functions and procedures (reading a string is modified by the interactive flag) for both terminal and disk devices. The interactive attribute has no effect when writing to a device.

When the buffer variable is used an assumption is made regarding data direction depending on where the buffer variable is used. If the buffer variable is used on the left side of an assignment statement then it is assumed that the data will be written to the device. If the buffer variable is used on the right side of an assignment statement (as part of an expression) then before using the data in the element buffer, the buffer is filled. This is accomplished by executing the get procedure if the element buffer is not valid.

Most users will not need to define their own custom devices and so all of the details of what a device descriptor and device driver are will be left until chapter 10, for those that need this information. The standard I/O devices (input, output, auxout, and keyboard) are pre-defined and are covered below.

Files are declared by following the reserved words "file" "of" by the type of the file, or by the word "text". Note that "file of text" may be abbreviated to just "text" for compatibility with the ISO standard. In the standard versions of the compiler a file will use the disk management in the operating system. A file declaration may also have the words "interactive" and "packed" preceding the word "file". As is true with all types, packed has no meaning in this compiler and is ignored. If interactive is specified it will affect the meaning of the EOF and EOLN flags as described above. Interactive is not normally used for files.

file-type = [**interactive**] [**packed**] **file of** (type | **text**) | **text**



STANDARD DEVICES

The operation of the standard devices depends on which operating system you will be using. Chapter 17 contains detailed information on each of the devices, including the operation and modes for files. The following information is general in nature and is aimed at non-OS-9 users, OS-9 users should consult chapter 17 before using the standard devices input or auxout.

The standard I/O device "input" is normally attached to the system terminal's keyboard, with buffering done by the operating system similar in action to the command line buffering. This device is capable of input transfers only. Data typed from the terminal is also echoed in the same manner as command line buffering. No actual data is available from this device until a carriage return is entered. The normal character delete and line delete edit functions are available from the terminal, others may be available depending on the operating system.

The compiler B option (as described in chapter 1) will affect the operation of the input device. If the B option is off and the operating system defined break key is hit, then control will return to the operating system when the break function is executed. If the B option is on and the break key is hit, then control will not return to the operating system but will return true from the break function. This is used when cleanup is required before returning to the operating system, or when you wish to use the break key for some other purpose. The standard input device is implicitly declared as : interactive device of text.

The standard I/O device "output" is normally attached to the system terminal's screen. This device is capable of output transfers only. The standard output device is implicitly declared as : device of text.

The standard I/O device "auxout" is normally attached to the system printer. This device is capable of output transfers only. The standard auxout device is implicitly declared as : device of text.

The standard I/O device "keyboard" is normally attached to the system terminal's keyboard. This device is capable of input transfers only. No buffering is done by this device and there are no special character recognized. This device does not echo its input to the screen. The standard keyboard device is implicitly declared as : interactive device of char.

POINTERS

In ISO standard Pascal pointers are only used to dynamically create variables on a stack-like structure called a heap. Pointers types are declared by following the caret symbol "^" by the base type of the pointer. NOTE: The caret symbol ^ is also referred to as a circumflex, or in fact may be an up-arrow on some terminals or printers. This symbol will be referred to as a caret in this manual.

The base type of the pointer may be any of the simple types (boolean, integer, hex, character, longinteger, or real) or may be a user defined type identifier.

The actual pointer variable is compatible with any other pointer variable and with any hex value. The hex value of $0 is used to signify that the pointer does not point to a valid object and the reserved word "nil" is used to represent this $0 value. It is therefore not a good idea to start the heap at location zero.

To access the data object that the pointer points to, follow the pointer name with the caret. For example :

```
pntr = ^real ;   { pntr points to real values }

pntr := nil ;    { the pointer now points to nothing }

new (pntr) ;     { allocates 4 bytes for a real on the stack
                   and puts the address for it in pntr       }

pntr^ := 4.3 ;   { stores 4.3 at the location pointed to by pntr }
```

The procedure "new" stores the next available location on the heap into its parameter. It also moves the internal heap pointer up by the size of the type that the pointer points to.

OmegaSoft Pascal allows a number of extensions to make pointers even more powerful. A pointer is essentially a hex value with a special attribute, meaning that any operation that can be used for a hex variable will work for a pointer. For example, suppose you wanted to store strings on the heap. The pointer declaration would be :

```
line : string [80] ;
pntr : ^line ;
```

To put the string 'ABC' on the heap you could use :

```
line := 'ABC' ;
new (pntr) ;
pntr^ := line ;
```

But this would use up 81 bytes of heap space for the 3 byte string, for which only 4 bytes are needed. OmegaSoft Pascal allows an optional parameter to the new procedure which overrides the default size of the base type. Therefore we can specify how much heap space we really need :

```
new (pntr : length(line) + 1) ;
pntr^ := line ;
```

The integer expression following the colon in the new procedure specifies the number of bytes needed off of the heap. To move from one string to another pointer arithmetic may be used :

```
pntr := pntr + hexint(length(pntr^) + 1) ;
```

This extension can save a great deal of space when dealing with variable length structures.

```
pointer-type = ^ type
pointer-constant = nil | hex-constant
```

LONGHEX

Longhex numbers were added to the compiler so that the 32 bit addressing of the 68000 could be supported in OmegaSoft cross software. We hope that some of you will find a use for them.

Longhex variables are stored as four bytes, with the most significant byte being at the lowest address. When longhex values are used in expressions, passed as parameters by value, or returned from a function, they occupy four bytes on the stack with the most significant byte being at the lowest address.

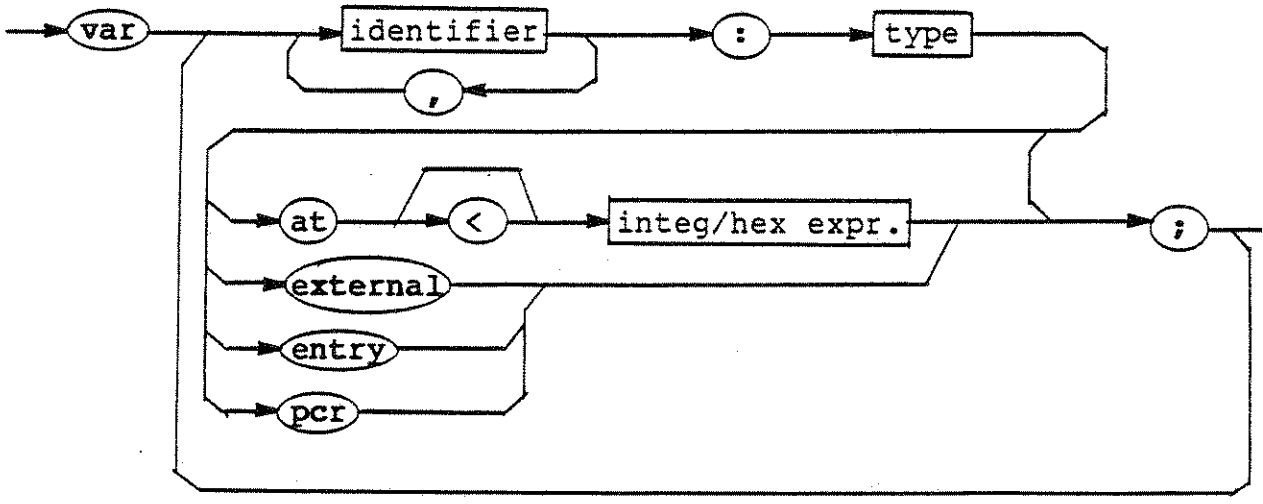
```
longhex-type = longhex
longinteger-constant = longhex-number |
                       longhex-constant-identifier
```

VARIABLE DECLARATIONS

Variables are normally allocated on the data stack. The first variable declared is allocated immediately below the stack frame, and subsequent variables below that. Since variables are referenced relative to the stack frame, small and often used variables should be allocated first. This will allow those variables to use shorter and faster addressing instructions to access them.

The variable section must not exceed 32757 bytes in length or else a compile-time error will occur.

```
variable-declaration = var var-definer { var-definer }
var-definer = identifier {, identifier} : type [(pcr | entry |
                external | at [ < ] address-expression) ;
address-expression = integer-constant-expression |
                     hex-constant-expression
```



EXTENDED ADDRESSING

If the type in the variable declaration is followed with the word "at" and then an integer or hex constant expression then the variable will start at the indicated address. The variable will be accessed using the 6809 extended addressing mode. This mode is useful for placing a variable at a specific location in memory - such as a byte variable for an ACIA or PIA, or an array starting at the start of a RAM or ROM.

DIRECT PAGE ADDRESSING

If the type in the variable declaration is followed with the word "at", the symbol "<", and then an integer or hex constant expression then the variable will start at the indicated address in base page. The variable will be accessed using the 6809 direct page addressing mode. This mode is useful for common variables between Pascal and assembly language or as way to obtain very fast access of critical global variables.

PCR ADDRESSING

If the type in the variable declaration is followed with the word "pcr" then the variable will be accessed using the 6809 program counter relative addressing mode. An external reference (XREF) will be issued for the variable using its declared name truncated to 6 characters and upshifted.

As an example :

```
initarray : array [1..16] of byte pcr ;
```

will generate : XREF INITAR

and to access the array will use : LEAX INITAR,PCR

This mode is useful for placing tables or other "constant variables" in along with the program, normally by using assembly language and linking it with the Pascal program.

EXTERNAL ADDRESSING

If the type in the variable declaration is followed by the word "external" then it means that the variable is allocated on the stack but declared in another module. An external reference (XREF) will be generated using the variable name truncated to 6 characters and upshifted. As an example :

```
flag : byte external ;
```

will generate : XREF FLAG

and to access the array will use :

```
 LEAX FLAG,Y   { if at lexical level 1 (global)}
or
 LDX -6,Y
 LEAX FLAG,X   { if not at lexical level 1 }
```

This mode is useful for referencing global variables from the main program (with those defined as entry) from a module. This is an alternative to declaring global variables in every module that uses them and is very attractive when only a few variables need be imported from the main program into the module.

ENTRY ADDRESSING

If the the type in the variable declaration is followed by the word "entry" then normal stack addressing will be used, but in addition an external definition will be generated for the variable. The external definition (XDEF) will use the variable name truncated to 6 characters and upshifted. As an example :

```
flag : byte entry ;
```

```
will generate :  XDEF FLAG
                 FLAG EQU $xxxx
```

where xxxx is the stack offset to be added to the global stack frame pointer.

This mode is useful for defining global variables in the main program for use in a module with the variable defined as external. This mode can also be used when you need to access a global variable from an assembly language routine by inserting into the code :

```
 XREF FLAG
   .
   .
 LDX -6,Y
 LEAX FLAG,X
```

The entry allocation attribute is automatically applied to the standard I/O devices if declared in the program parameter list.

NOTE : These special addressing modes may only be used in the global variable declaration section, not within procedures.

SCOPE OF IDENTIFIERS

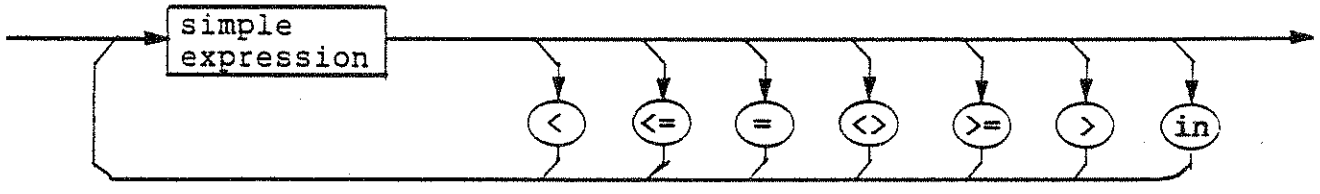
Identifiers declared in a block are accessible from any inner blocks unless redefined in the inner block. Outer blocks may not use identifiers declared in inner blocks. The innermost definition is used in cases where the same identifier is used to define different objects in nested blocks. These scoping rules apply to all identifiers used as names for constants, variables, types, procedures, and functions.

EXPRESSIONS

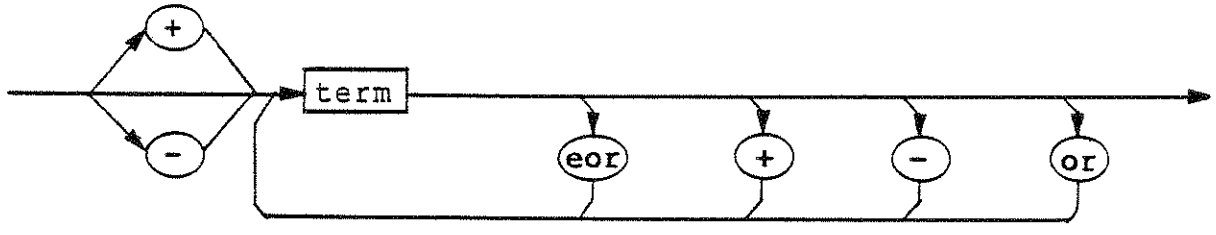
Expressions are a combination of one or more constants, variables, or functions separated by operators.

```
expression = simple-expression {(< | <= | = | <> | >= | > | in)
             simple-expression }
simple-expression = [(+ | -)] term {(eor | + | - | or) term}
term = factor {(* | / | ** | div | mod | and | << | >>) factor }
factor = (unsigned-constant | variable | ( expression ) |
         function-call | not factor | set-constructor)
variable = (variable-identifier | field-identifier) {( ^ |
           [ expression {, expression} ] | . field-identifier)}
function-call = function-identifier [( expression
                {, expression } ) ]
set-constructor = [ [ set-element {, set-element } ] ]
set-element = expression [.. expression]
```

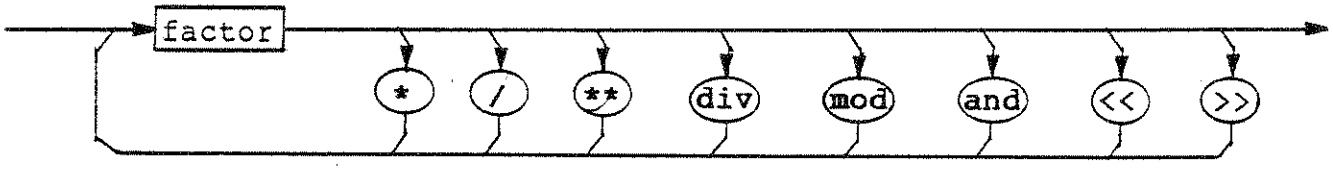
Expression :



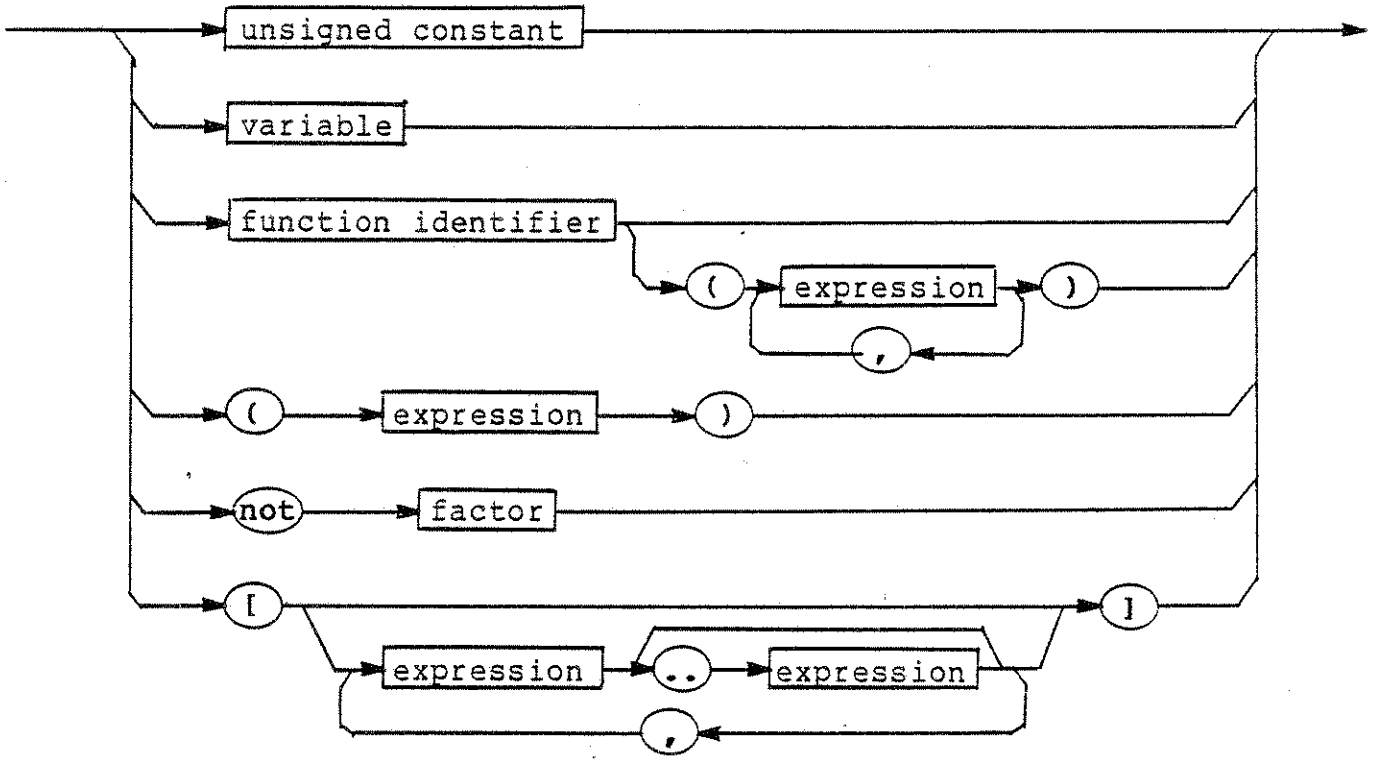
Simple expression :



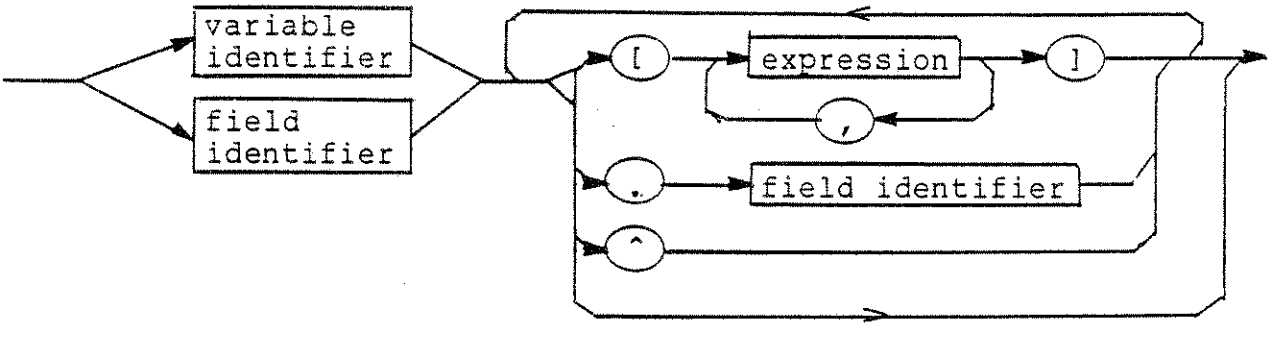
Term :



Factor :



Variable :



The parts of an expression are evaluated Factor first, term, simple expression, and expression last; and within a part it is evaluated left to right. For all binary operators (except IN), the left and right expression parts must be of the same type, with the following automatic conversions taking place:

Expression part type		Converted type
integer	-->	real
longinteger	-->	real
integer	-->	longinteger
hex	-->	longhex

Binary operator compatibility table :

	Expression type									
Operator	Bool	Char	Int	Hex	Lhex	Lint	Real	Set	Str	Enum
<= = <> >=	x	x	x	x	x	x	x	x	x	x
< >	x	x	x	x	x	x	x		x	x
in	L	L	L	L				R		L
div mod		x	x	x	x	x				
*		x	x	x	x	x	x	x		
** /							x			
and or eor	x	x	x	x	x	x				
<< >>		x	x	x	x	x				
+ -		x	x	x	x	x	x	x		
not	x	x	x	x	x	x				

Notes :
L = Left operand.
R = Right operand.
x = both operands.
Unary + or - are defined for integers, longintegers and reals.
Pointers are allowed the same operations as hex.

ARITHMETIC EXPRESSIONS

Operator	Example	Meaning
+	a + b	add a to b
-	a - b	subtract b from a
*	a * b	multiply a times b
**	a ** b	raise a to the b power
/	a / b	divide a by b
div	a div b	divide a by b and truncate
mod	a div b	remainder after dividing a by b
and	a and b	bitwise anding of a and b
or	a or b	bitwise oring of a and b
eor	a eor b	bitwise eoring of a and b
not	not a	bitwise complement of a
<<	a << b	shift a left b places (fill with 0)
>>	a >> b	shift a right b places (fill with 0)

The addition and subtraction operators (+ and -) work on character, integer, longinteger, hex, longhex, and real operands. Addition or subtraction of character or hex operands cannot generate an error (values wrap around 0). Addition or subtraction of integer values cannot generate an error unless the compiler S option is enabled.

The multiplication operator (*) will work on character, integer, longinteger, hex, longhex, and real operands. Multiplication of character operands cannot generate an error unless the compiler S option is enabled.

The exponentiation and division operators (** and /) work on real operands only. If either operand is an integer or longinteger, they will be converted to real before being processed. An attempted division by zero will yield an error if range checks are on, else a garbage result. The base operand of an exponentiation must not be negative.

The div and mod operators work on character, integer, longinteger, hex, and longhex operands. The left and right operands must be of the same type and the right operand must not be zero.

The and, or, eor, and not operators work on boolean, character, integer, longinteger, hex, and longhex operands. The left and right operands must be of the same type. All are bitwise except the not operator for boolean : 0 becomes 1 and 1 becomes 0.

The shift left and shift right (<< and >>) operators work on character, integer, longinteger, hex, and longhex operands. The left and right operands must be of the same type. The right operand is the shift count and if negative will reverse the direction of the shift (a << b = a >> -b). Regardless of the direction of the shift, zeroes are shifted in.

Except where otherwise noted above, the arithmetic operators allow you to mix integer, longinteger, and real operands. In the case of a mis-match the smaller type is converted automatically to the larger type.

RELATIONAL EXPRESSIONS

The relational operators generate a boolean result based on a comparison of arithmetic or boolean operands.

Operator	Example	Meaning
=	a = b	a is equal to b
<	a < b	a is less than b
>	a > b	a is greater than b
<=	a <= b	a is less than or equal to b
<>	a <> b	a is not equal to b
>=	a >= b	a is greater than or equal to b

The relational operators work on boolean, character, integer, longinteger, hex, longhex, real, string, array, and record. String comparisons are done on a character by character basis using the ASCII ordering. If two strings have identical characters up until one of the strings runs out of characters, the shorter string is considered smaller.

SET EXPRESSIONS

Operator	Example	Meaning
+	a + b	union of a and b, result of all elements in either a or b
-	a - b	difference of a and b, result of all elements in a but not in b.
*	a * b	intersection of a and b, result of all elements in both a and b
=	a = b	a is equal to b
<>	a <> b	a is not equal to b
<=	a <= b	elements in a are also in b
>=	a >= b	elements in b are also in a
in	k in a	ordinal k is in the set a

Operators +, -, and * require that both operands be sets and the result type will be a set. Operators =, <>, <=, and >= require that both operands be sets and the result type will be boolean. The in operator requires that the left operand be an ordinal and that the right operand be a set, returning a boolean result.

PRECEDENCE OF OPERATORS

The precedence of operators in Pascal is based on which syntax (expression, simple-expression, term, or factor) that the operator is located in. Within any precedence level evaluation is done from left to right. Below is a table of precedences going from highest to lowest :

syntax section	operators	precedence
factor	( ) not	1
term	* / ** div mod and << >>	2
simple-expression	+ - eor or	3
expression	< <= = <> >= > in	4

VARIABLES

Variables may be either whole variables, or some part of the variable. Records may have individual fields accessed by following the record name with a period and the field name. Arrays may have individual elements accessed by indexing (index within brackets). Strings may be indexed by either an integer or byte expression in brackets, the result being a character. Devices and pointers may have their element or base type accessed by following the device or pointer name with a caret (^).

These access methods may be nested in complex structures, for instance in :

```
type
  a = string [40] ;
  b = ^a ;
  c = record
        d : array [1..5] of b ;
        e : record
              f : integer ;
              g : real
            end ;
        h : a
      end ;
var
  i : array [#1..#20] of c ;
  j : file of c ;
```

The following accesses can be made :

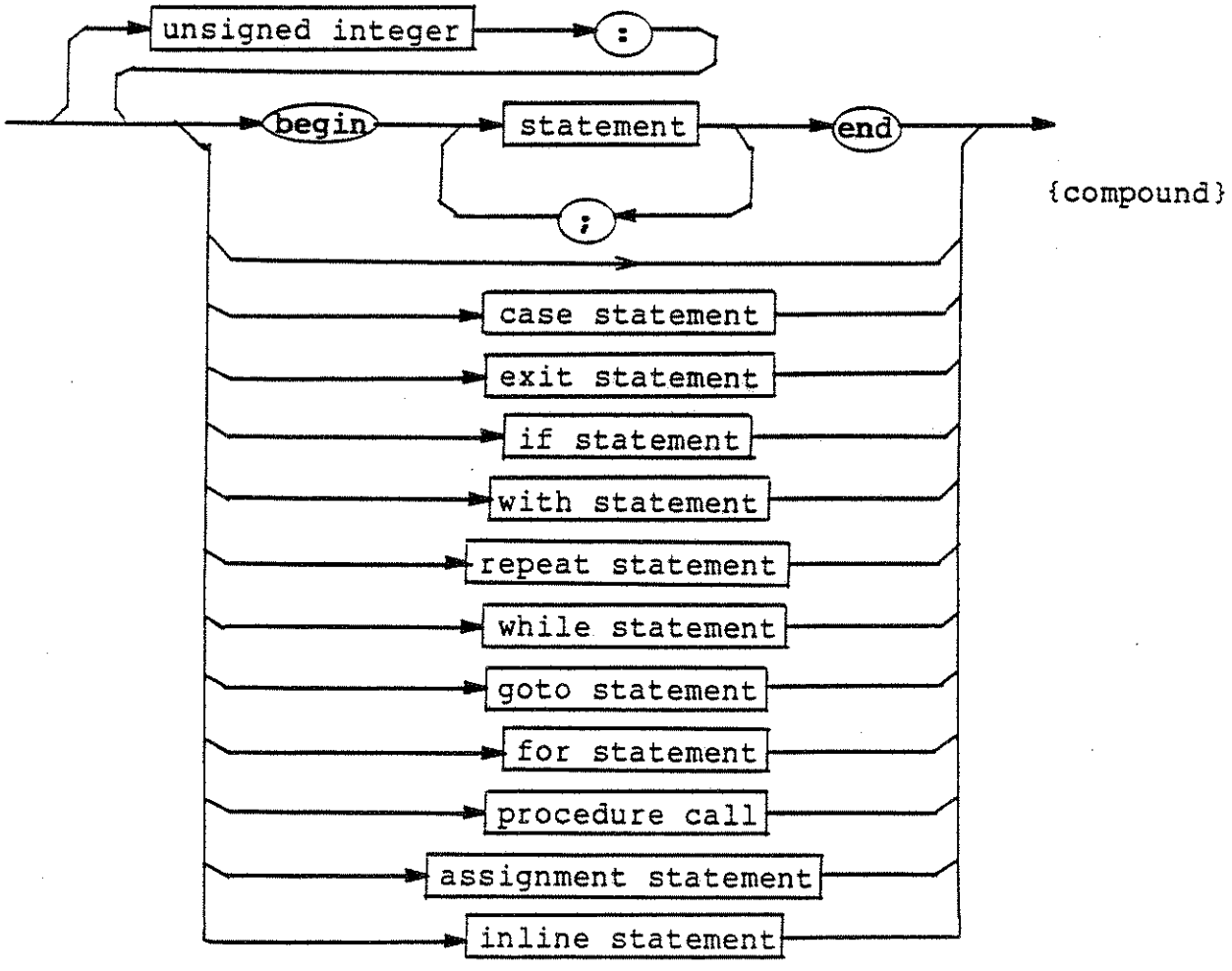
```
  i[#5].h           is a string [40]
  i[#10].e.f        is an integer
  j^.d[3]^          is a string [40]
```

STATEMENTS

Statements are the part of the Pascal program that actually do the work required. Statements may only appear within the begin..end of the program block or of a procedure or function block. Remember that semicolons are used as statement separators in Pascal, not terminators.

```
statement = [ unsigned-integer : ] (null-statement |
            compound-statement | assignment-statement |
            case-statement | if-then-else-statement |
            for-statement | repeat-statement | while-statement |
            exit-statement | goto-statement | with-statement |
            procedure-call | inline-statement)
```

statement :



The null-statement consists of nothing and useful in case statements to do nothing if a certain case is met, it is also found when extra semicolons are used, as in :

```
begin
  writeln ('help') ;
end
```

A null statement is found between the semicolon and the end, and will do no harm.

COMPOUND STATEMENT

The compound statement allows grouping of Pascal statements to act as one statement. The sequence :

```
begin
  k := 5 ;
  j := -k
end
```

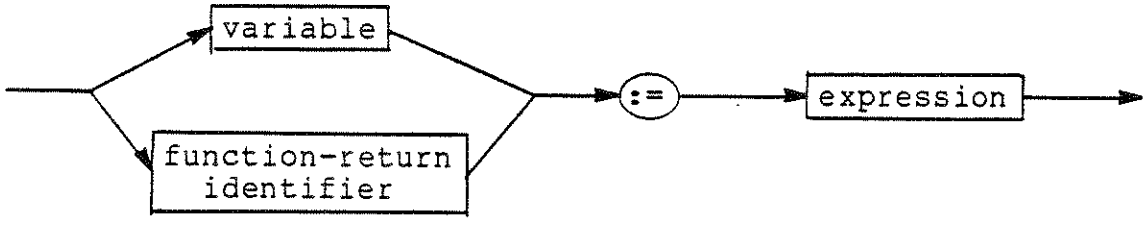
Can now represent one statement. A compound statement may contain any other kind of Pascal statement within it. The begin .. end section of a block is essentially a compound statement that must be present.

compound-statement = **begin** statement {; statement} **end**

ASSIGNMENT STATEMENT

The assignment statements allows the result of an expression (covered in chapter 3) to be placed into a variable or function return variable.

assignment-statement = (variable | function-return) := expression



The following table lists the expression types that are compatible (can be assigned) to the various variable types :

variable type	acceptable expression
boolean	boolean
character	character
enumerated	enumerated (any)
integer	integer
hex	hex
subrange	subrange (same base type), base type
longinteger	longinteger or integer
longhex	longhex or hex
real	real, longinteger, or integer
string	string, character, array*
array	array (same size), character*, string*
record	record (same size)
set	set (any)
pointer	pointer or hex

NOTE : The acceptable expressions marked with an asterisk "*" may only be used if the array is less than 127 bytes.

CONDITIONAL STATEMENTS

Conditional statements select a statement to execute based on the value of an expression.

CASE STATEMENT

The case statement selects zero or one statement to execute based on the value of a selector expression. The expression is compared against constants until a match is found, at which point the statement that goes with the matching constant is executed. The expression may be boolean, character, enumerated, integer, hex, or string. Constants of the same type as the expression are used to determine which statement to execute. More than one constant may be specified to select a statement and except for string selectors, a subrange of constants may be used to compare against.

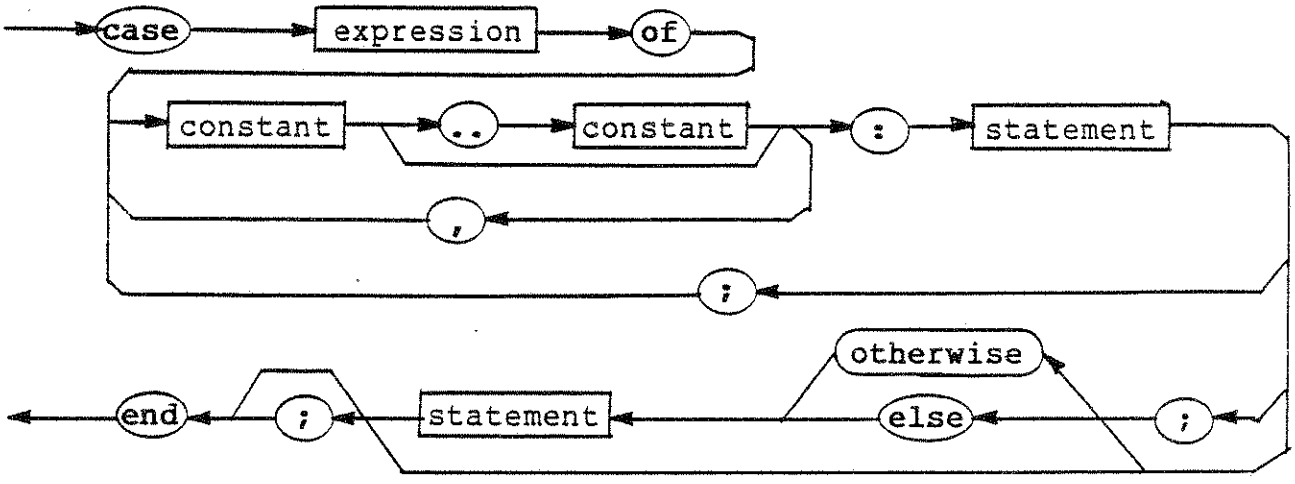
If none of the constants match, then none of the statements will be executed. If it is desired that a statement be executed when there is no match then an optional "else" or "otherwise" clause may be inserted at the end of the case statement. There is no check for duplication of case constants in this compiler, the comparisons are done in the order of declaration.

For example the following code will pick out carriage returns and line feeds separately from all other control characters :

```
case ch of
  #$D : {process carriage return} ;
  #$A : {process line feed} ;
  #$0..#$1F : {process all other control characters}
  else {process non-control characters}
end ;
```

Note that boolean and character selectors are the fastest, followed by integer and hex, with string being the slowest.

```
case-statement = case expression of case-selection [;] [(else |
                 otherwise) statement [;]] end
case-selection = case-constant-list : statement
                 {; case-constant-list : statement}
case-constant-list = constant [.. constant]
                     {, constant [.. constant]}
```



IF-THEN-ELSE STATEMENT

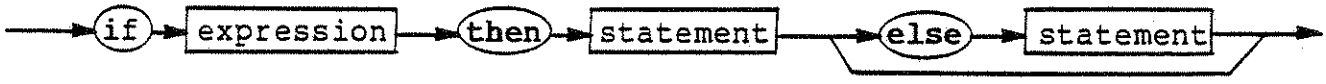
The if-then-else statement allows execution of the statement following the "then" if the boolean expression following the "if" is true. It will execute the statement following the "else" if the boolean expression is false. The else clause may be left off, leaving you with an if-then statement which will only execute the statement if the boolean expression is true. Note that in nested if-then-else statements an else always goes with the closest unpaired then, such that in :

```
if expr1 then if expr2 then stat1 else stat2 ;
```

is equivalent to and should be formatted as :

```
if expr1
  then
    if expr2
      then
        stat1
      else
        stat2 {goes with second if} ;
```

if-statement = **if** boolean-expression **then** statement
[**else** statement]



REPETITIVE STATEMENTS

The repetitive statements are used for forming loops to execute statements multiple times until a condition is met.

FOR STATEMENT

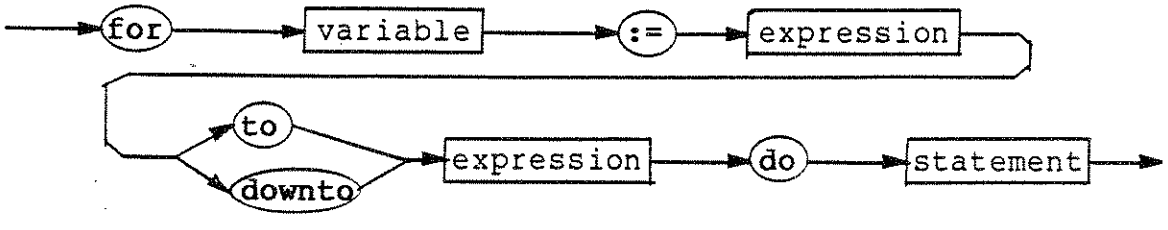
There are two forms of the for statement, the first in which one is added to the control variable each time through the loop. This form is specified by following the first expression with the word "to". The value of the first expressions is assigned to the control variable. This value is then compared against the second expression, and if it is less than or equal will execute the statement. After the execution of the statement the control variable is compared against the second expression and if equal, the for statement is terminated. If the control variable is not equal then one is added to it and the statement is executed again. This process repeats until the control variable is equal to the second expression.

The second form is where one is subtracted from the control variable. This form is specified by following the first expression with the word "downto". The value of the first expression is assigned to the control variable. This value is then compared against the second expression, and if it is greater than or equal will execute the statement. After the execution of the statement the control variable is compared against the second expression and if equal, the for statement is terminated. If the control variable is not equal then one is subtracted from it and the statement is executed again. This process repeats until the control variable is equal to the second expression.

In either case if the statement is never executed then the control variable will have the value of the first expression. If the for statement finishes normally then the control variable will have the value of the second expression.

The control variable and expression type may be boolean, character, integer, enumerated, or hex type.

for-statement = **for** variable := expression (**to** | **downto**)
expression **do** statement

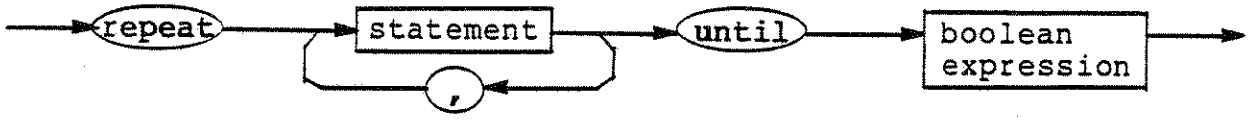


REPEAT STATEMENT

This statement will execute one or more statements until a condition is met. The statement(s) will be executed until the boolean expression is true. An endless loop can be created by making the expression equal to the boolean constant "false", in this case an unconditional branch will be used in the code generated.

Note that the statement(s) will always be executed at least once since the test is done at the end of the loop.

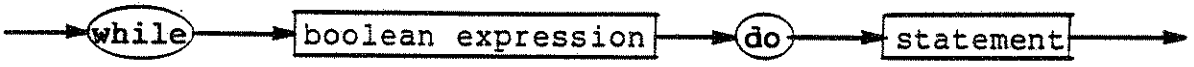
repeat-statement = repeat statement {; statement}
until expression



WHILE STATEMENT

The while statement is similar to the repeat statement except the condition is checked at the top of the loop. If the boolean expression is true, then the statement is executed and the loop repeated. This will continue until the boolean expression is false. An infinite while loop is constructed by using the boolean constant "true" as the expression. The statement will not be executed at all if the expression is false when the while statement is entered.

while-statement = **while** expression **do** statement



TRANSFER STATEMENTS

The transfer statements are used to modify the flow of control in a program where one of the standard conditional or repetitive statements is not suitable.

EXIT

The exit statement is used to "jump out of" loops. It can only be used to exit a for, repeat, or while loop. If executed inside a loop it will jump to the point at which the loop would terminate normally. For example :

```
repeat
  statement1 ;
  statement2 ;
  if condition
    then
      exit ;
  statement3
until condition ;
statement              {this is the destination for the exit}
```

If the optional integer parameter in parenthesis is used, then that will be the loop count. If you are nested 2 deep in loops then you can exit both loops by using : exit (2) . You cannot exit more levels of loops than you are currently in and you cannot use this to exit a block.

The exit statement is usually referred to as a "structured goto" and is preferable to a goto unless ISO standard code is desired.

exit-statement = **exit** [(unsigned-integer-constant)]

GOTO

The goto is rarely required and should be avoided when possible. The OmegaSoft Pascal compiler will not allow you to jump in or out of a block and you cannot jump into certain statements including whiles, for loops, and case statements using a string selector. The destination of a goto must be a labeled statement (covered later in this chapter) and there is a limit of 8 forward defined goto's in a block. A forward defined goto is a goto to a label that follows the goto.

goto = **goto** unsigned-integer

WITH STATEMENT

The with statement is used to abbreviate the notation required for accessing fields in a record. In the with statement a record variable name is specified and when the statement is being compiled any identifiers will be checked to see if they are a field of the specified record before checking the rest of the symbol table. In the record :

```
applicant : record
              name : string [40] ;
              age : integer ;
              address : array [1..2] of string [40]
              score : integer
            end ;
```

The fields could be setup as :

```
applicant.name := 'Joe Smith' ;
applicant.age := 26 ;
applicant.address[1] := '4500 Nonroad st.' ;
applicant.address[2] := 'Reseda, NJ 90009' ;
applicant.score := 93
```

Or as :

```
with applicant do
  begin
    name := 'Joe Smith' ;
    age := 26 ;
    address[1] := '4500 Nonroad st.' ;
    address[2] := 'Reseda, NJ 90009' ;
    score := 93
  end
```

Note that using a with statement affects the code generated to access each field. In the first example each field was accessed individually from the stack mark. In the second example at the start of the with statement the record was accessed from the stack mark and that address is pushed on the data stack. The fields are accessed as offsets from this value saved on the stack. In this example it might generate more code to use the with statement than without it. Where a with statement can save code is when the record access involves array indexing or file access. If the data structure allows you to have an array of records like :

```
symbol = array [1..1000] of record
                              name : string [6] ;
                              address : hex ;
                                  .
                                  .
                            end ;
```

Then it would be advantages to use a with like :

```
with symbol [j] do
  begin
    name := curr_name ;
    address := location ;
      .
      .
  end ;
```

Since the array access would be done only once for many field accesses.

More than one record name can appear in a with statement and in this case :

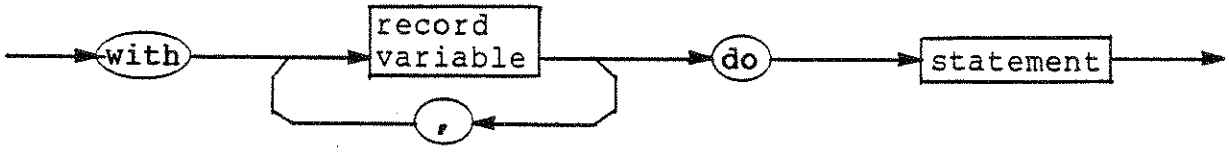
```
with name1, name2, name3 do .....
```

is equivalent to :

```
with name1 do
  with name2 do
    with name3 do ...
```

and the innermost name (name3) would be searched first when looking for possible fields.

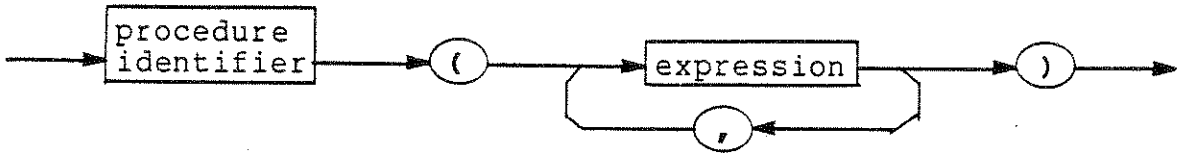
with-statement = **with** variable {, variable} **do** statement



PROCEDURE CALL

A procedure call is used to pass actual parameters to a procedure and execute it. See chapter 5 for a description of procedures and parameters. The parameters are pushed onto the data stack before calling the procedure.

procedure-call = procedure-identifier [(expression
{, expression})]



LABELED STATEMENTS

Labeled statements are used as destinations for goto statements. The unsigned-integer must have already been declared as a label in this block and any gotos to the label must also be in this block.

labeled-statement = unsigned-integer : statement

INLINE STATEMENT

The inline statement allows assembly language source code to be passed through the compiler unchanged. Note this feature should be avoided and that the assembly language code will be ignored by the debugger. This feature will only work if the output of the compiler is to be assembled and linked.

The inline statement is specified by following a exclamation mark "!" by the code to be passed through. Everything up until the end of the line is passed through. A group of these lines represent one statement. If the last character on the line (before the carriage return) is a semicolon then the semicolon is not passed through but is instead considered a statement separator. As an example :

```
begin
  ! LDA #4
  !LOOP TST $A00C
  ! BMI LOOP
  ! STA $A00D ;
  a := true
end ;
```

PROCEDURES AND FUNCTIONS

Procedures and functions are program units that a similar to the main program block. They may have parameters that are passed on the data stack, they may have all of the declaration sections that a program can have, and they have an execution section.

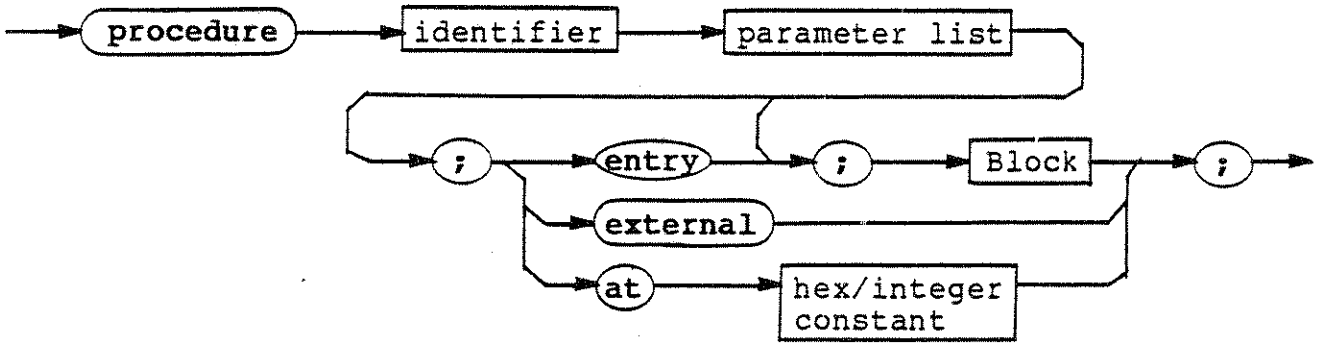
Procedures and functions are used for two reasons. They allow you to build a routine that can be called many different places and thus reduce redundant code. They also allow you to isolate related code and variables into a separate section, reducing the probability of undesired side-effects and allowing re-usability of code.

Procedures and functions are similar in nature and the term "procedure" in this chapter is meant to refer to either a procedure or function. Where there is a difference between the two, it will be pointed out.

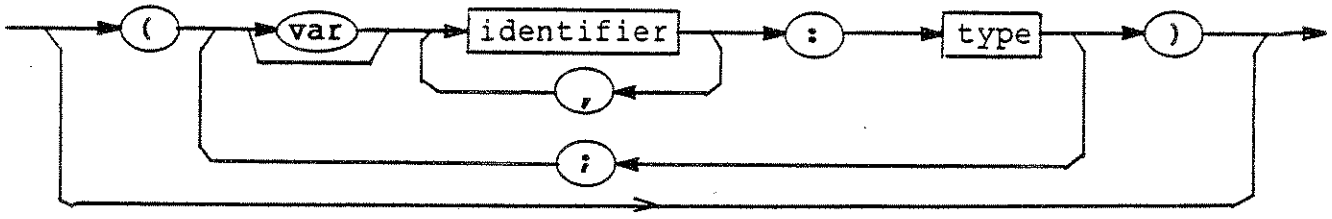
There are many predeclared procedures included with the compiler, these will be covered in the next two chapters. This chapter will deal only with user written procedures.

FORMAT OF A PROCEDURE OR FUNCTION

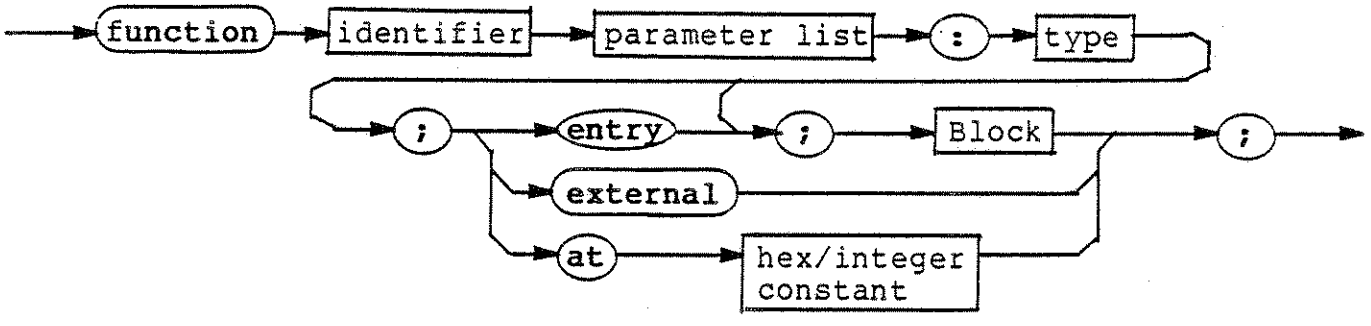
```
procedure-declaration = procedure identifier parameter-list ;
            (block | entry ; block | external | at address) ;
parameter-list = ( | (parameter-element {; parameter-element}) )
parameter-element = [var] identifier {, identifier} : type
address = integer-constant-expression | hex-constant-expression
procedure-call = procedure-identifier [( (expression | variable)
                 {,(expression | variable)} )]
```



Parameter list:



function-declaration = function identifier parameter-list : type
; (block | **entry** ; block | **external** | **at** address)



PARAMETERS

FORMAL PARAMETER LIST

The formal parameter list specifies what parameters are to be passed to the procedure from the caller and how they are to be passed ; by value or by address (variable). The identifiers in the parameter list and local variables in the procedure are allocated at the time of the procedure call. At the end of the procedure execution they disappear (de-allocated). Variables defined global to a procedure may also be used by the procedure.

VALUE PARAMETERS

A value parameter has its value copied to a local variable created by the procedure at the time of the procedure call. Any changes made to this parameter within the procedure does not affect the original variable passed. This is evident by the fact that an expression may be passed as a value parameter. Devices may not be passed by value.

VARIABLE PARAMETERS

A variable parameter (definition in parameter list prefixed by a "var") actually has its address passed to the procedure at the time of the procedure call. Therefore, any changes made to the parameter within the procedure does change the original variable passed. Only a variable (not an expression) may be passed as a variable parameter.

Variable parameters are essentially pointers to values and therefore can also be expressed as passing pointers by value. The two sections of code that follow are similar in operation :

```
program variable_parameter ;
  var
    x : integer ;
  procedure a (var y : integer) ;
    begin
      write (y)
    end ;
  begin
    a (x)
  end.

program pointer_parameter ;
  var
    x : ^integer ;
  procedure a (y : ^integer) ;
    begin
      write (y^)
    end ;
  begin
    a (x)
  end.
```

There is one important difference between the above two sections of code, type checking. Using variable parameters allows the compiler to check the type of the actual parameter against the parameter declaration. Since any pointer is compatible with any other pointer, there is no type checking and a real could have been passed just as easy as an integer. This can be used where such tricks are desired - such as processing a record as an array but it also can get you into big trouble if you are not careful.

FUNCTION RETURN TYPE

The function return type may be one of the following: integer, longinteger, hex, character, enumerated, boolean, real, string, set, or subrange. The return value may also be an array or record as long as it does not exceed 126 bytes in size. The function name must be assigned to from within the function, this sets the function return value.

TYPE COMPATIBILITY

When a procedure call is compiled the actual parameters used are compared against the procedure declaration to verify that the types are correct. The following table lists what actual parameters can be passed to what declarations :

declaration type	acceptable parameter
boolean	boolean
character	character
enumerated	enumerated (any)
integer	integer
hex	hex
subrange	subrange (same base type)
longinteger	longinteger or integer*
longhex	longhex or hex*
real	real, longinteger*, or integer*
string	string or character*
array	array (same size - see dynamic arrays)
record	record (same size)
set	set (any)
devices and files	devices and files (variable only)
pointer	pointer and hex

NOTE : The acceptable parameters marked with an asterisk "*" may only be used if they are to be passed by value, not by variable.

DYNAMIC ARRAY VARIABLES

If an array is passed by variable then the array passed may be the same size or smaller than the declaration. This feature may be used to create procedures capable of handling different size arrays as parameters. Refer to OmegaSoft Application Note number 5 for details on this feature. Note that this is not the method specified for "conformant array parameters" specified in the ISO standard for compliance level one.

SIDE EFFECTS

A side effect is the modification of a non-local variable in a procedure. This may take the form of an assignment to a non-local variable or as an assignment to a parameter that is declared as a variable parameter. Side effects should be avoided since their use will cause a procedure to modify its environment in ways that are difficult to document. Some side effects may cause non-portable code to be produced, as in :

```
program test (input,output) ;
  var
    x : integer ;

  function b (y, z : integer) ;
    begin
      b := z + x * y ;
      x := y
    end ;

  begin
    read (x) ;
    a := b (b(3,5),b(2,7)) ;
    writeln (a)
  end.
```

In the OmegaSoft compiler procedure parameters are evaluated left to right but this is left to the implementor in the ISO standard. If you were to transport this code to a compiler that evaluates right to left then the result would be different.

DECLARATION OPTIONS

If none of the declaration options are used then the procedure block must immediately follow the declaration part and the procedure name will not be known outside of this compilation. The Declaration options provide for mutually recursive routines, assembly language procedures, and modular compilation.

FORWARD

Since every procedure must be declared before it is referenced a problem is created when two or more procedures call each other recursively. The forward declaration is included in the language to get around this problem :

```
procedure a (j : integer) ; forward ;

procedure b (k : integer) ;
  begin
    a (k)
  end ;

procedure a ;
  begin
    b (j)
  end ;
```

Note that in the forward declaration the full parameter list is used but that when the procedure block is going to be presented that only the procedure name is used.

There is an area where the OmegaSoft compiler differs from the ISO standard for forward declarations. The variables in the parameter list for the procedure declared forward remain known at that lexical level until the end of the block for that procedure is presented. This means that you could not have :

```
procedure a (j : integer) ; forward ;
procedure b (j : integer) ;    { multiply defined variable }
```

Since forwards are rarely needed this difference should not be a serious problem. Note that the use of forward declared procedures also eats up symbol table space faster than normal in the compiler.

EXTERNAL

If a procedure is declared as external then its name is used truncated to 6 characters and upshifted. In the declaration :

```
procedure setports (b : byte) ; external ;
```

the following code is emitted :

```
 XREF SETPOR
```

and if the procedure is called then the following code will be used :

```
 LBSR SETPOR
```

This is used to access procedures defined in other Pascal modules or to access assembly language procedures.

ENTRY

If a procedure has the word "entry" placed between the declaration and its block then the procedure name truncated to 6 characters and upshifted will be made available as an entry point. In the declaration :

```
procedure setports (b : byte) ; entry ;
  begin
    .
    .
  end ;
```

the following code is emitted at the start of the procedure :

```
 XDEF SETPOR
SETPOR EQU *
```

Using this declaration you can reference this procedure in another Pascal module by declaring it external, or you can call the procedure from assembly language by using a :

```
 XREF SETPOR
  .
  .
 LBSR SETPOR
```

ABSOLUTE

This option is similar in use to the external option but instead of using the name of the procedure, an absolute (extended addressing) address is provided. For the procedure :

```
procedure setpor (b : byte) ; at $F31A ;
```

The following code would be generated to call it :

```
 JSR $F31A
```

This is useful for placing commonly used Pascal or assembly language routines at fixed addresses, such as in an EPROM, and calling them from Pascal programs.

PREDECLARED FUNCTIONS

OmegaSoft Pascal features a wide range of predeclared functions, both ISO standard and extensions. The predeclared functions will be presented by category, and within a category, alphabetically.

ARITHMETIC FUNCTIONS

ABS

Returns the absolute value of its parameter. The result type will be the same type as the parameter and will be positive regardless of the sign of the parameter.

```
abs (5) = 5    abs (-5) = 5
```

```
abs-function = abs ( (integer-expression | real-expression |
                      longinteger-expression) )
```

ARCCOS

Returns the arc cosine (in radians) of its parameter. The result type will be real. The parameter may be in the range of -1 to 1 inclusive, any parameter outside of this range will generate an overflow error at runtime.

```
arccos-function = arccos ( (integer-expression | real-expression
                           | longinteger-expression) )
```

ARCSIN

Returns the arc sine (in radians) of its parameter. The result type will be real. The parameter may be in the range of -1 to 1 inclusive, any parameter outside of this range will generate an overflow error at runtime.

```
arcsin-function = arcsin ( (integer-expression | real-expression
                           | longinteger-expression) )
```

ARCTAN

Returns the arc tangent (in radians) of its parameter. The result type will be real. The parameter may be in the range of -2E9 to 2E9 inclusive, any parameter outside of this range will generate an overflow error at runtime.

```
arctan-function = arctan ( (integer-expression | real-expression
                           | longinteger-expression) )
```

COS

Returns the cosine of its parameter. The result type will be real and in radians. The parameter may be in the range of -51000 to 51000 inclusive, any parameter outside of this range will generate an overflow error at runtime.

```
cos-function = cos ( (integer-expression | real-expression
                    | longinteger-expression) )
```

EXP

Returns the value of E raised to the power of the parameter. The result type will be real. The exp function is equivalent to e ** parameter with e being equal to 2.718282 . The parameter must in the range of -31 to 31, any parameter outside of this range will generate an overflow error at runtime.

```
exp-function = exp ( (integer-expression | real-expression
                    | longinteger-expression) )
```

LN

Returns the natural logarithm (base e) of the parameter. The result type will be real. The parameter must not be zero or negative or else it will generate an invalid argument for square root or log error at runtime.

```
ln-function = ln ( (integer-expression | real-expression
                  | longinteger-expression) )
```

LOG

Returns the common logarithm (base 10) of the parameter. The result type will be real. The parameter must not be zero or negative or else it will generate an invalid argument for square root or log error at runtime.

```
log-function = log ( (integer-expression | real-expression
                    | longinteger-expression) )
```

RANDOM

Returns a pseudo-random real value between (but not including) 0.0 and 1.0 . The parameter must be a real variable identifier and will be used a 32 bit shift register with feedback. The parameter is first shifted, the upper 9 bits are masked off to make the number between 0.5 and 1.0, 0.5 is then subtracted to get the number between 0.0 and 0.5, the number is then multiplied by 2 to get the final result between 0.0 and 1.0 .

The parameter should be initialized to a non-zero number before use, as zero is the only forbidden state for the "seed". The initialization number should have a good distribution of 1's and 0's to provide a even distribution, 0.5 is a very bad choice (only 1 bit on) while non powers of two like pi or e are good choices.

random-function = **random** (real-variable)

SIN

Returns the sine of its parameter. The result type will be real and in radians. The parameter may be in the range of -51000 to 51000 inclusive, any parameter outside of this range will generate an overflow error at runtime.

sin-function = **sin** ((integer-expression | real-expression
| longinteger-expression))

SQR

Returns the square of its parameter. The result type will be the same type as the parameter. This function is equivalent to parameter * parameter and will generate less code and execute in approximately the same amount of time (logs are not used).

sqr-function = **sqr** ((integer-expression | real-expression
| longinteger-expression))

SQRT

Returns the square root of the parameter. The result type will be real. The parameter must not be zero or negative or else it will generate an invalid argument for square root or log error at runtime.

sqrt-function = **sqrt** ((integer-expression | real-expression
| longinteger-expression))

TAN

Returns the tangent of its parameter. The result type will be real and in radians. The parameter may be in the range of -51000 to 51000 inclusive, any parameter outside of this range will generate an overflow error at runtime.

tan-function = **tan** ((integer-expression | real-expression
| longinteger-expression))

TYPE CONVERSION FUNCTIONS

The OmegaSoft Pascal compiler features an orthogonal method of converting between the nine basic data types : boolean, enumerated type, character, integer, longinteger, hex, longhex, real, and string. The method is to use the desired type as the function name, or the standard identifier "enum" for enumerated type.

In general there is only one parameter. When converting from real to anything other than real or string a second parameter may be used. The second parameter may be "round", "trunc", or "floor" and if not used then rounding will be used. Other cases where a second parameter is allowed will be pointed out where applicable. If you convert from one type to the same type, no code is generated. Many other instances also will generate no code, only changes in the type of expression, these cases occur if the type of parameter is not mentioned in the description for the general type conversion functions (boolean, enum, char, integer, hex, longhex, longinteger, real, and string).

```
conv-parm = (boolean-expression | enumerated-type-expression |
             character-expression | integer-expression |
             hex-expression | longhex-expression |
             longinteger-expression | string-expression)

real-round = real-expression {, (round | trunc | floor) }
```

BOOLEAN

Converts its parameter into a boolean value. For enumerated type, character, integer, hex, longhex, and longinteger parameters the least significant byte is simply "anded" with one and is equivalent to the odd function. For a real parameter it is first converted to an integer and then anded. When using a string parameter it must not have any leading spaces and the first letter must be a Y, T, N, or F (or lower case equivalents) or else a conversion error will be generated. If the first letter is a Y or T then boolean true will be returned, if the first letter is a N or F then a boolean false will be returned.

```
boolean-function = boolean ( (conv-parm | real-round) )
```

CHAR

Converts its parameter into a character. For longhex and longinteger parameters the 4 byte value is removed from the stack, with only the least significant byte being retained. For a real parameter the value is converted into an integer and only the least significant byte is retained.

For a string parameter there is a second parameter allowed. The second parameter must be a byte or integer constant and specifies which byte of the string is to be returned as a character.

If no second parameter is used then the first data byte of the string will be returned. Using a second parameter of zero will return the dynamic length.

char-function = **char** (conv-parm {, (character-constant | integer-constant)})

CHR

Returns a character (byte) equivalent of its parameter. This is done by removing the most significant 8 bits of its parameter. If the compiler S option is on then checks will be generated to verify that the most significant 8 bits were zero. If range checks are on and they are not zero, then a truncation error will occur at runtime.

chr-function = **chr** ((integer-expression | hex-expression))

ENUM

Converts its parameter into an enumerated type value. For longhex and longinteger parameters the 4 byte value is removed from the stack, with only the least significant byte being retained. For a real parameter the value is converted into an integer and only the least significant byte is retained. A string parameter is converted into an integer (see integer function with string parameter) and the least significant byte retained.

FLOOR

Will return the integer equivalent of its parameter. The integer result will be the largest integer less than the real value. If the parameter cannot be represented as an integer a truncation error will be generated at runtime.

floor (4.6) = 4 floor (-4.6) = -5

floor-function = **floor** (real-expression)

HEX

Will return the hex equivalent of its parameter. If the parameter is boolean, enumerated, or character, then a zero most significant byte will be added. If the parameter is longhex or longinteger then the 4 byte value is pulled from the stack and the least significant 2 bytes retained. If the parameter is real then the value will be converted to an integer and this result treated as a hex number (hex(-10000.0) would return $D8F0). With a string parameter it must not have any leading or trailing spaces or else a conversion error will be generated. The digits 0-9 and the letters A-F (or a-f) are acceptable for hexidecimal input. If the value exceeds $FFFF then a conversion error occurs.

hex-function = **hex** ((conv-parm | real-round))

INTEGER

Will return the integer equivalent of its parameter. If the parameter is boolean, enumerated, or character, then a most significant byte will be added that is the sign extension of the original byte. If the parameter is longhex or longinteger then the 4 byte value is pulled from the stack and the least significant 2 bytes retained. If the parameter is real then the value will be converted to an integer. With a string parameter it must not have any leading or trailing spaces or else a conversion error will be generated. The digits 0-9 and a leading plus or minus are acceptable for decimal input. If the value exceeds plus or minus maxint then a conversion error occurs.

integer-function = **integer** ((conv-parm | real-round))

LONGHEX

Will return the longhex equivalent of its parameter. If the parameter is boolean, enumerated, or character, then 3 zero bytes will be added as the most significant and pushed on the stack. If the parameter is integer or hex then 2 zero bytes will be added as the most significant and pushed on the stack. If the parameter is real then the value will be converted to a longinteger similar to round, trunc, or floor. This value will then be treated as a longhex value (longhex(-3437138.0) will be $FFCB8DAE). With a string parameter it must not have any leading or trailing spaces or else a conversion error will be generated. The digits 0-9 and the letters A-F (or a-f) are acceptable for hexidecimal input. If the value exceeds $FFFFFFFF then a conversion error occurs.

longhex-function = **longhex** ((conv-parm | real-round))

LONGINTEGER

Will return the longinteger equivalent of its parameter. If the parameter is boolean, enumerated, or character, then 3 bytes will be added as the most significant reflecting the sign extension of the original byte, and pushed on the stack. If the parameter is integer or hex then 2 bytes will be added as the most significant reflecting the sign extension of the original most significant byte, and pushed on the stack. If the parameter is real then the value will be converted to a longinteger similar to round, trunc, or floor. With a string parameter it must not have any leading or trailing spaces or else a conversion error will be generated. The digits 0-9 and a leading plus or minus are acceptable for decimal input. If the value exceeds plus or minus maxlint then a conversion error occurs.

longinteger-function = **longinteger** ((conv-parm | real-round))

ODD

Returns a boolean result that represents the least significant bit of its parameter. This tells you whether the parameter is odd (ls bit one) or even (ls bit zero).

```
odd-function = odd ( (character-expression | integer-expression |
                      hex-expression | longinteger-expression |
                      longhex-expression) )
```

ORD

Returns the integer equivalent of its parameter. If the parameter is one byte (boolean, enumerated, character) it will simply add a zero most-significant byte. If the parameter is integer or hex then no code is generated.

```
ord-function = ord ( (boolean-expression | character-expression |
                      enumerated-expression | hex-expression |
                      integer-expression) )
```

REAL

Will return the real equivalent of its parameter. If the parameter is boolean, enumerated, or character, then it will be sign extended to an integer, and then converted to real. If the parameter is integer, hex, longhex, or longinteger, it will be converted to real (signed conversion). With a string parameter it must not have any leading or trailing spaces or else a conversion error will be generated. The digits 0-9, a leading plus or minus, and an E or e for the exponent are acceptable for floating point input. If the value exceeds 5E-19 or 5E18 then a conversion error occurs.

```
real-function = real ( (conv-parm | real-expression) )
```

ROUND

Will return the integer equivalent of its parameter. The integer result will be the closest integer to the real value. If the parameter cannot be represented as an integer a truncation error will be generated at runtime.

```
round (4.6) = 5      round (-4.6) = -5

round-function = round ( real-expression )
```

STRING

Returns the ascii string equivalent of its parameter. The string returned is in the same format as when the values are written to a text file (see write in chapter 7) with no fieldwidth (no leading or trailing blanks). If the parameter is enumerated then its integer equivalent is converted to a string. If the parameter is character, it is converted to a string of length one. If the parameter is of type real then there is an additional byte or integer parameter to be used as the precision (same function as in write procedure).

```
string-function = string ( (conv-parm | real-expression
                                    [, ( character-expression |
                                       integer-expression)]) )
```

TRUNC

Will return the integer equivalent of its parameter. The integer result will be the real value with its fractional part set to zero. If the parameter cannot be represented as an integer a truncation error will be generated at runtime.

```
trunc (4.6) = 4      trunc (-4.6) = -4
```

```
trunc-function = trunc ( real-expression )
```

I/O AND RUNTIME STATUS FUNCTIONS

BREAK

Returns boolean true if the device parameter has encountered a break condition. Break is normally used as an operator input. In standard OmegaSoft Pascal supplied device drivers break is used as an operator wait or abort. The break status is cleared after the break function is executed. See chapter 17 for detailed information on the break function for your operating system.

If the parameter is not included then the standard input device is used as the parameter.

break-function = **break** [(device-variable)]

CONVERSION

Returns boolean true if the last ascii to internal format conversion encountered an error. These conversions occur during the boolean, integer, hex, longinteger, longhex, and real functions with a string parameter and the read and readln procedures when reading boolean, integer, hex, longhex, longinteger, or real values from a text device.

This value is cleared after any successful conversion takes place. This function is only useful when conversion error checking (compiler toggle C) is off, since if it is on, a conversion error will halt the program execution.

This function may be used in interactive programs to avoid crashing the program if the operator enters an invalid value.

conversion-function = **conversion**

DEVERR

Returns the device parameter's error byte as a character (byte). Refer to chapter 10 for a description of this byte and chapter 17 for the operating-system dependant value it may take on. This function is only useful when I/O error checking (compiler toggle I) is off, since if it is on, an I/O error will halt the program execution.

This function may be used in interactive programs to avoid crashing the program if the operator enters an invalid file name or if the program requires that it check for or delete files that may not be in the directory.

If the parameter is not used then the standard input device will be used as the parameter.

deverr-function = **deverr** [(device-variable)]

EOF

Returns boolean true if the device parameter has hit end of file. The meaning of eof is system and device dependant and chapter 17 should be referenced for more information.

If the device is not interactive the element buffer must be valid and if not valid a get procedure will be executed before checking for end of file. If the device is interactive then the current contents of the element buffer will be used, valid or not.

If the device is a text device and eof is true then the element buffer (device^) will be a space.

If the parameter is not used then the standard input device will be used as the parameter.

eof-function = **eof** [(device-variable)]

EOLN

Returns boolean true if the text device parameter is currently on end of line. If eoln is true then the element buffer (device^) will be a space.

If the device is not interactive the element buffer must be valid and if not valid a get procedure will be executed before checking for end of line. If the device is interactive then the current contents of the element buffer will be used, valid or not.

If the parameter is not used then the standard input device will be used as the parameter.

eoln-function = **eoln** [(device-variable)]

MEMAVAIL

Returns a hex result which is the amount of free space available on the heap. This may be used in interactive programs to make sure that sufficient room is left on the heap for the anticipated data before using the new procedure and risking a heap overflow runtime error. This function can also be used to allocate maximum sized buffers on the heap by using memavail (minus some slop) as the extra parameter to the new procedure.

The amount of space left on the heap is affected by heap allocation (new procedure) and heap de-allocation (release procedure).

If you are running your program with a single section of RAM for heap and data stack then procedure and function calls will also affect the amount left. If this is the case then you should leave enough slop to handle the deepest procedure/function calling when you calculate the amount of heap than can be used. In the single ram area setup, procedures and functions use up heap space because their parameters, stack frames, and local variables move the data stack pointer down towards the heap pointer.

memavail-function = **memavail**

RANGE

Returns boolean true if a range error was generated (errors 4 through 12) since the last time this flag was cleared. The range error is different in that it is not cleared when a successful operation takes place, it must be cleared by the programmer.

The range error flag is cleared by an assignment :

```
range := false ;
```

In this way it is more of a standard variable rather than a function. This function is only useful when range error checking (compiler toggle R) is off, since if it is on, a range error will halt the program execution.

range-function = **range**
range-assignment = **range** := boolean expression

STRING FUNCTIONS

CLINE

Returns a string with the contents of the command line (running under operating system only). If no parameter is provided or if the parameter has a value of zero then the entire command line will be returned. If the parameter is a positive number "n" then the "nth" command line argument will be returned. If there is no argument "n" then a null string will be returned. Command line arguments are separated by spaces or commas.

If the parameter is equal to -3 then the character "<" and the characters that follow it will be returned. If there is no "<" character then a null string will be returned. In a similar manner -2 will look for ">" and -1 will look for ">>".

```
cline-function = cline [( (integer-expression |
                           character-expression) )]
```

CONCAT

Returns a string that is the concatenation of its parameters. The parameters will be concatenated in the order in which they are listed. If the resulting string would exceed 126 characters the concatenation is not performed and a dynamic length error will be generated.

```
concat-function = concat ( param {, param} )
param = (character-expression | string-expression)
```

INDEX

Returns an integer that corresponds to the location that one string is contained within another. The result is the location (starting character number) in the first parameter where the second parameter occurs. If there is no occurrence then 0 will be returned.

```
index-function = index ( param , param )
param = (character-expression | string-expression)
```

LENGTH

Returns an integer in the range of 0 to 126 that represents the current dynamic length of a string parameter. This function is similar to :

```
ord (parameter[0])
```

The difference is that the length function will accept an expression and that it runs much slower. If you are trying to save time and space and need the length of a string parameter you can use the above equivalent. If you just need a byte representation of the length you should use :

```
string-variable [0]
```

rather than :

```
chr (length(string-variable))
```

```
length-function = length ( string-variable )
```

SUBSTR

Returns a string that is a subrange of the string parameter. The subrange is defined by a starting expression which is the first character to include and by a count expression which is the number of characters to include. If the sum of the starting and count expressions would exceed 126 or if the start expression is 0 then an error occurs. If the starting expression is past the end of the string then a null string will be returned. If the sum of the starting and count expressions is past the end of the string no error will be generated, the result string will just be shorter than the count expression specifies.

```
substr-function = substr ( string-param , start-expression ,
                                           count-expression )
string-param = (character-expression | string-expression)
start-expression = count-expression = integer-expression
```

UPSHIFT

Returns a character or string with each character that lies in the range "a" .. "z" converted to lie in the range "A" .. "Z". This function will return the same type as its parameter.

```
upshift-function = upshift ( (string-expression |
                              character-expression) )
```

MISCELLANEOUS FUNCTIONS

ADDR

Returns the hex absolute address of a variable. This is useful for doing all sorts of devious things with pointers. For instance, to access the device descriptor for the standard output device :

```
type
  descriptor = record
                 mode : byte ;
                 err : byte ;
                 drv : hex ;
                 elnt : integer ;
                 elmt : char ;
                 path : byte {os-9 only}
               end ;
var
  ptr : ^descriptor ;
begin
  ptr := addr (output) ;
  write ('Path number is : ',ord(ptr^.path):1)
   .
   .
```

Will write out the current path number (os-9 only).

addr-function = **addr** (variable)

PRED

Will return the value of its parameter decremented by one. The return type will be the same as the parameter type. For example :

```
type
  colors = (red, blue, green) ;
var
  a : byte ;
  b : integer ;
  c : colors ;
  d : boolean ;
begin
  a := pred (#5) ; { result is #4 }
  b := pred (5) ; {result is 4 }
  c := pred (green) ; { result is blue }
  d := pred (true) ; { result is false }
  .
  .
```

This function cannot generate an overflow, the one byte or two byte value will simply wrap around zero.

```
pred-function = pred ( (boolean-expression |
                        hex-expression |
                        integer-expression |
                        enumerated-expression |
                        character-expression) )
```

SIZEOF

Returns the integer size (in bytes) of its type or variable parameter. If the type contains a variant record the size returned will be the size of the largest variant.

```
sizeof-function = sizeof ( (type-identifier | variable) )
```

SUCC

Will return the value of its parameter incremented by one. The return type will be the same as the parameter type. For example :

```
type
  colors = (red, blue, green) ;
var
  a : byte ;
  b : integer ;
  c : colors ;
  d : boolean ;
begin
  a := succ (#5) ; { result is #6 }
  b := succ (5) ; {result is 6 }
  c := succ (red) ; { result is blue }
  d := succ (false) ; { result is true }
  .
  .
```

This function cannot generate an overflow, the one byte or two byte value will simply wrap around zero.

```
succ-function = succ ( (boolean-expression |
                        hex-expression |
                        integer-expression |
                        enumerated-expression |
                        character-expression) )
```

PREDECLARED PROCEDURES

OmegaSoft Pascal features a wide range of predeclared procedures, both ISO standard and extensions. The predeclared procedures will be presented by category, and within a category, alphabetically.

I/O PROCEDURES

CLOSE

Will send a signal code to the device driver to terminate use of that device, or in the case of a disk file will do an operating system close to the file that was open. No further data transfers may take place to the device until it is re-opened. This procedure is normally ignored by non-disk devices.

If any files or non-standard devices are opened in your program, you should close them before exiting your program rather than relying on the operating system to do it.

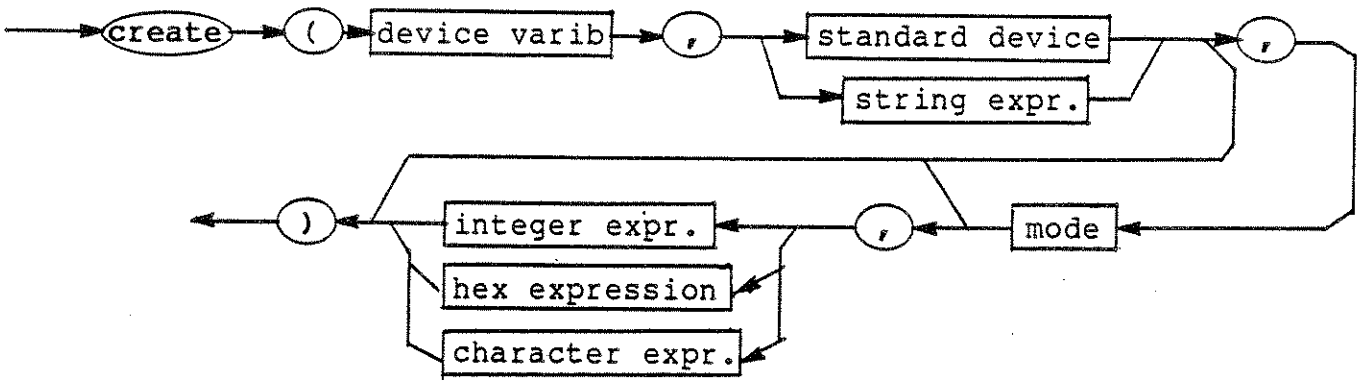
close-procedure = **close** (device-variable)

CREATE

This procedure is normally used to open a new file on a disk (file variable type). In this usage the string expression is the name to be given to the file and the mode is "input", "output", or "update" which determines the allowable data direction. The attribute expression (truncated to one byte if hex or integer) will be passed to the runtime code and is used as an operating system dependant flag (see chapter 17). It will be defaulted to 0 if not included. If the file already exists it will be truncated to zero length (may be deleted first depending on operating system).

If the second parameter is one of the standard device names "input", "output", "keyboard", or "auxout" then the device is initialized to be that device. In this case the last two parameters are not used. This is very useful for determining the nature of the device during runtime.

```
create-procedure = create ( device-variable ,
                   (standard-device | string-expression) [, mode
                   [, attribute-expression ]] )
standard-device = (input | output | keyboard | auxout)
mode = (input | output | update)
```



DEL

This procedure is normally used on disk-file devices to delete the file specified by the string expression. If the file does not exist, an error will occur.

del-procedure = **del** (device-variable , string-expression)

DEVINIT

If a file or device is declared as part of an array or record then the compiler will not automatically initialize the device at the start of the block. This procedure is used for that purpose. As an example, in the following declaration :

```
var
  x : integer ;
  fgroup : array [1..5] of record
                             status : boolean ;
                             fyle : text
                           end ;
```

The following code could be used to initialize each file in the array :

```
  for x := 1 to 5 do
     devinit (fgroup[x].fyle) ;
```

devinit-procedure = **devinit** (device-variable)

GET

Will transfer data from a device into its element buffer (device^). If the eof status is set on the device then performing a get will generate an eof runtime error. If eof is not set then get will attempt to transfer data from the device into the element buffer. If eof is hit during this transfer then the eof status is set and the element buffer is undefined (space for text devices). If any other error is encountered during the transfer it will generate a runtime error. This procedure will also set the eoln status if appropriate on text devices.

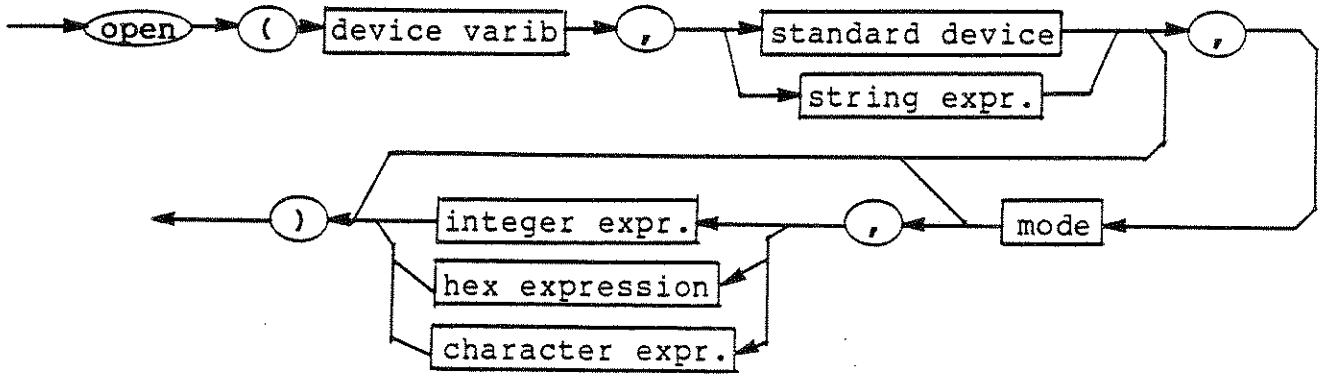
get-procedure = **get** (device-variable)

OPEN

This procedure is normally used to open an existing file on a disk (file variable type). In this usage the string expression is the name to be given to the file and the mode is "input", "output", or "update" which determines the allowable data direction. The attribute expression (truncated to one byte if hex or integer) will be passed to the runtime code and is used as an operating system dependant flag (see chapter 17). It will be defaulted to 0 if not included. If the file does not exist a runtime error will be generated.

If the second parameter is one of the standard device names "input", "output", "keyboard", or "auxout" then the device is initialized to be that device. In this case the last two parameters are not used. This is very useful for determining the nature of the device during runtime.

```
open-procedure = open ( device-variable ,
                 (standard-device | string-expression) [, mode
                 [, attribute-expression ]] )
standard-device = (input | output | keyboard | auxout)
mode = (input | output | update)
```



PAGE

Will bring the text device to top of form if supported by the device driver. If the device is not currently at the beginning of the line then a writeln will be performed before issuing the top of form. See chapter 17 for more information on the page operation for the standard devices. In most cases this procedure is only used when driving a printer. If the parameter is not used then the standard output device will be used as the parameter.

page-procedure = **page** [(device-variable)]

PUT

Will transfer data from the device's element buffer (device^) to the actual device. If any errors are encountered during the transfer then a runtime error will be generated.

put-procedure = **put** (device-variable)

READ AND READLN

These two procedures transfer data from a device to one or more variables. Readln is identical to read except it can only be used on text devices and will skip data up until the next end of line marker if necessary, thereby making sure that the next read or readln will start at the beginning of a new line.

For non-text devices the ISO standard specifies that read is equivalent to :

```
variable := device^ ;
get (device)
```

But in OmegaSoft Pascal a method is used to handle both interactive files and random access which requires a flag be used to indicate whether or not the contents of the element buffer is valid. Using this method a read is equivalent to :

```
if device^ is not valid
  then
    get (device) ;
variable := device^ ;
device^ := not valid
```

This will rarely make any difference in the operation of a standard ISO program. The only affect of this method is that data is not transferred into the element buffer until actually needed. The exception to this is when an eof or eoln function is performed on a non-interactive device which would do a get to check for those devices - this would set device^ valid so that a read procedure that follows would not do the initial get.

If the device is of type text then internal conversions are done using the boolean, integer, hex, longhex, longinteger, and real functions for boolean and numeric values using a string parameter. The read procedure will skip leading blanks when reading these types (this includes the blank that represents end of line) until it finds the end of a non-blank sequence. If a character is being read then nothing will be skipped and a space will be read in place of the end of line mark. If a string is read then it will read all characters up to but not including the end of line mark. Note that if a string read is attempted while eoln is true then a null string will be returned and eoln will still be true.

The sequence :

```
readln (device, var1, var2, var3)
```

is equivalent to :

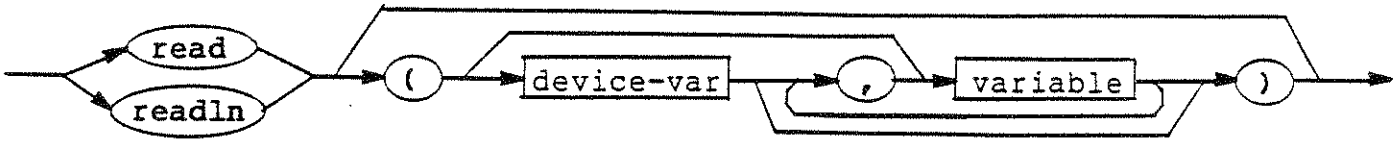
```
read (device, var1)
read (device, var2)
readln (device, var3)
```

and so it makes no sense to read multiple strings in this manner since the read of var1 will leave you on the eoln mark and so var2 and var3 must receive a null string. Multiple string reads must be done with multiple readln procedures - one per string.

If the eof status for the devices is true when a read is called then a runtime error will be generated.

If the first parameter is not a device variable then the standard input device will be used.

```
read-procedure = read [( (device-variable [{, variable}] |
                          variable {, variable}) )]
readln-procedure = readln [( (device-variable [{, variable}] |
                             variable {, variable}) )]
```



RESET

There are two forms of the reset procedure. The first one includes only a device parameter and is used to open a file for input using a file name derived from the command line. As an example :

```
program test (input,output,file1,file2) ;
  var
    file1 : text ;
    file2 : file of integer ;
  begin
    reset (file1) ;
    reset (file2) ;
```

file1 will be opened using the first command line parameter and file2 will be opened using the second command line parameter. This is a result of the declaration of the two device variables in the program parameter list. In this case reset (file1) is equivalent to reset (file1, cline(1)) (second form below). If this form is used and the device name is not in the program parameter list then a null string will be used for the file name.

The second form of reset uses an additional string parameter which specifies the file name to use. This second parameter will override the default file name if the device is declared in the program parameter list. reset (device, name) is equivalent to open (device, name, input).

reset-procedure = **reset** (device-variable [, string-expression])

REWRITE

There are two forms of the rewrite procedure. The first one includes only a device parameter and is used to create a new file for output using a file name derived from the command line. As an example :

```
program test (input,output,file1,file2) ;
  var
    file1 : text ;
    file2 : file of integer ;
  begin
    rewrite (file1) ;
    rewrite (file2) ;
```

file1 will be created using the first command line parameter and file2 will be created using the second command line parameter. This is a result of the declaration of the two device variables in the program parameter list. In this case rewrite (file1) is equivalent to rewrite (file1, cline(1)) (second form below). If this form is used and the device name is not in the program parameter list then a null string will be used for the file name.

The second form of rewrite uses an additional string parameter which specifies the file name to use. This second parameter will override the default file name if the device is declared in the program parameter list. rewrite (device, name) is equivalent to create (device, name, output).

```
rewrite-procedure = rewrite ( device-variable
                              [, string-expression] )
```

SEEK

Is normally used to access records on a disk file. The parameter is used as the record number to move to, this is multiplied by the size of the element to obtain a 32 bit byte offset into the file. After a seek is done either a read/get or write/put can be done and the file then acts sequentially until a new seek is done. The operation of the seek is very operating system dependant and is covered in chapter 17. The first record of a file is 0 and therefore seek (device,0) is equivalent to a rewind operation.

```
seek-procedure = seek ( device-variable , (integer-expression |
                        hex-expression | longinteger-expression |
                        longhex-expression) )
```

WRITE AND WRITELN

These two procedures transfer data from one or more expressions to a device. Writeln is identical to write except it can only be used on text devices and will add a carriage return (and possibly a line feed depending on the device) to the end of the data to move to the next line.

For non-text devices write is equivalent to :

```
device^ := expression ;
put (device)
```

If the device is of type text then internal conversions are done using the string function for boolean and numeric values. If the first parameter is not a device variable then the standard output device will be used.

Following each expression may be an optional colon and fieldwidth parameter whose functions depends on the data type to be written. If the fieldwidth is positive it will pad spaces on the left, if it is negative then it will pad spaces on the right. Only the least significant 8 bits of the parameter are used. We will present the formatting used on a type by type basis :

Boolean

If the fieldwidth is not specified then a value of 6 will be used. If the specified fieldwidth is not sufficient to hold the string then it will be truncated on the right.

```
Fieldwidth = 6   "  TRUE"    " FALSE"
Fieldwidth = 4   "TRUE"      "FALS"
Fieldwidth = 1   "T"         "F"
Fieldwidth = 0   ""          ""
Fieldwidth = -6  "TRUE  "    "FALSE "
```

Character

If the fieldwidth is not specified then a value of 1 will be used. If the specified fieldwidth is not sufficient (0) to hold the character then it will be truncated to zero length.

```
Fieldwidth = 6   "     A"
Fieldwidth = 1   "A"
Fieldwidth = 0   ""
Fieldwidth = -6  "A     "
```

Integer

If the fieldwidth is not specified then a value of 10 will be used. If the specified fieldwidth is not sufficient to hold the string then it will be expanded.

```
Fieldwidth = 6   "  1234"
Fieldwidth = 4   "1234"
Fieldwidth = 1   "1234"
Fieldwidth = -6  "1234  "
```

Hex

If the fieldwidth is not specified then a value of 6 will be used. If the specified fieldwidth is not sufficient to hold the string then it will be truncated on the right.

```
Fieldwidth = 6   "  F3A7"
Fieldwidth = 4   "F3A7"
Fieldwidth = 1   "F"
Fieldwidth = 0   ""
Fieldwidth = -6  "F3A7  "
```

Longhex

If the fieldwidth is not specified then a value of 10 will be used. If the specified fieldwidth is not sufficient to hold the string then it will be truncated on the right.

```
Fieldwidth = 10   "  F3A70481"
Fieldwidth =  4   "F3A7"
Fieldwidth =  1   "F"
Fieldwidth =  0   ""
Fieldwidth = -10  "F3A70481  "
```

Longinteger

If the fieldwidth is not specified then a value of 16 will be used. If the specified fieldwidth is not sufficient to hold the string then it will be expanded.

```
Fieldwidth = 10   "    431874"
Fieldwidth = 6    "431874"
Fieldwidth = 4    "431874"
Fieldwidth = -10  "431874    "
```

Real

If the fieldwidth is not specified then a value of 16 will be used. If the specified fieldwidth is not sufficient to hold the string then it will be expanded. There is another optional parameter than can follow the fieldwidth. If a colon and a parameter follows the fieldwidth it is the precision. If the precision is positive the format will be in fixed point notation and the parameter is the number of digits past the decimal point. If the precision is negative then the format will be floating point with exponent. In this floating point format there will always be 6 digits to the right of the decimal point and a 2 digit exponent. The default for the precision is negative.

```
Fieldwidth = 14 , Precision = -    "  3.141593E+00"
Fieldwidth = 14 , Precision = 3    "         3.141"
Fieldwidth = 14 , Precision = 6    "      3.141593"
Fieldwidth = 14 , Precision = 10   "  3.1415930000"
Fieldwidth = 5  , Precision = 3    " 3.141"
Fieldwidth = 5  , Precision = 6    "3.141593"
```

In the floating point format there is always one position used in front of the number to be used as a sign, space for plus, "-" for negative. In the fixed point format there is only a position used if the number is negative (for the - character).

```
Fieldwidth = -14, Precision = -    " 3.141593E+00 "
Fieldwidth = -14, Precision = -    "-3.141593E+00 "
Fieldwidth = -10, Precision = 5    "3.14159    "
Fieldwidth = -10, Precision = 5    "-3.14159   "
```

String

If the fieldwidth is not specified then a value equal to the dynamic length of the string will be used. If the specified fieldwidth is not sufficient to hold the string then it will be truncated on the right.

```
Fieldwidth = default "OmegaSoft"
Fieldwidth = 10      "OmegaSoft "
Fieldwidth = 5       "Omega"
Fieldwidth = 0       ""
Fieldwidth = -10     " OmegaSoft"
```

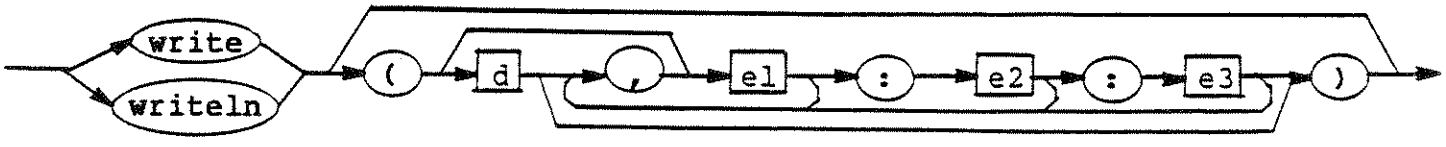
The sequence :

```
writeln (device, exprl, expr2, expr3)
```

is equivalent to :

```
write (device, exprl) ;
write (device, expr2) ;
writeln (device, expr3)
```

```
write-procedure = write [( (device-variable
                  [{,write-expression}] |
                  write-expression {,write-expression}) )]
writeln-procedure = writeln [( (device-variable
                    [{, write-expression }] |
                    write-expression {, write-expression  }) )]
write-expression = expression [: field [: precision ]]
field = precision = (integer-expression | character-expression)
```



where :
- d = device variable
- e1 = data expression
- e2 = fieldwidth
- e3 = precision

DYNAMIC VARIABLE MANAGEMENT PROCEDURES

These procedures are used to handle allocation of variables on the heap.

DISPOSE

This procedure will "disconnect" the pointer parameter from an the area of the heap it was pointing to. The pointer will have the value of nil after execution of this procedure. In the standard implementation of the heap this procedure does not actually give back any memory to the system.

If a more-advanced heap manager is used then the amount of memory to give back must be the same as the amount gotten using the new call. See the description of the new procedure on the use of the tag values and the size parameter.

```
dispose-procedure = dispose ( pointer-variable {, tag-values}
                              [: size ] )
```

MARK

This procedure will store the current position of the heap pointer into its parameter. This is the start of free storage in the heap. Further calls to new will cause the heap pointer to increase in magnitude. To effectively restore all of this memory used by the new procedure from the time the mark procedure was called, the release procedure would be used.

```
mark-procedure = mark ( (integer-variable | pointer-variable |
                         hex-variable) )
```

NEW

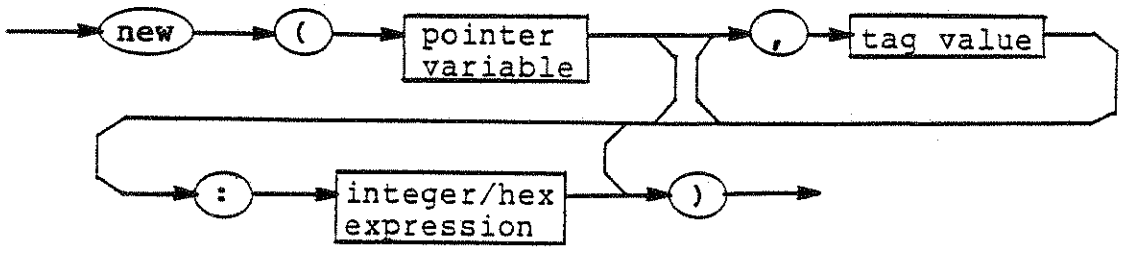
This procedure allocates storage on the heap and places the start of that storage in its pointer-parameter. The base type of the pointer variable will determine how much storage is allocated on the heap.

The optional tag values are field identifiers of records and in ISO Pascal are used to determine how much space to allocate based on which case variants are active in the record that the pointer points to. These tag values are ignored in OmegaSoft Pascal and the size of the largest variant is used.

An extension in OmegaSoft Pascal is the size parameter which allows you to override the default size with a specific value. This is especially useful when putting strings on the heap by using something like :

```
type
  lines = string [80] ;
var
  ptr : ^lines ;
  line : lines ;
begin
  new (ptr : length (line) + 1) ;
  ptr^ := line
```

```
new-procedure = new ( pointer-variable {, tag-field }
                      [ : size ] )
tag-field = identifier
size = (integer-expression | hex-expression)
```



RELEASE

Will store the value in its parameter into the heap pointer. This has the affect of de-allocating all dynamic variables allocated between the previous mark using the same parameter and this release.

```
release-procedure = release ( (pointer-variable | hex-variable |
                               integer-variable) )
```

MISCELLANEOUS PROCEDURES

HALT

Will pass the least significant byte of its parameter to the error handling software as an error. If the parameter is zero then it will return with no action. If the parameter is 255 ($FF) then it will stop the program with no error. Any other value for the parameter will signal a runtime error and stop the program.

```
halt-procedure = halt ( (character-expression |
                         integer-expression | hex-expression) )
```

MODULAR COMPILATION

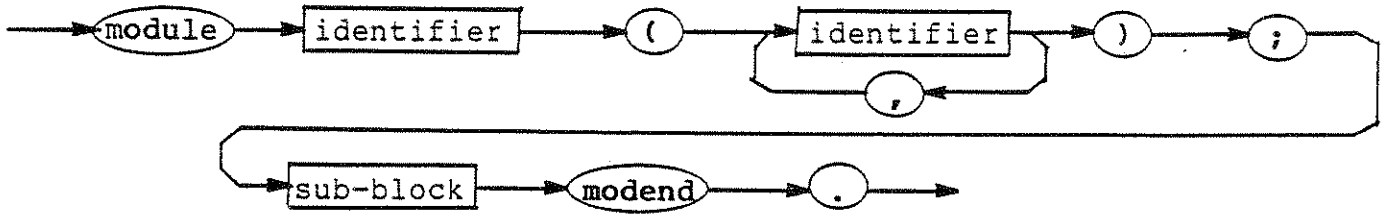
When a program becomes very large it will tend to delay development of the program. This results from longer editor load and save sequences, longer compilation time, and longer assembly times. If instead your program is broken up into one main program, and a number of modules, then only the portion being changed need be recompiled and reassembled. A typical method of modular programming would be to have one file be the main program with procedures and functions declared as external and only the main global block in the file. The actual procedures and functions would be declared as entry in the various modules.

To set up a modular system, first segment procedures and functions into modules, compile and assemble those modules, compile and assemble the main program, and then set up a linker control file to load first the setup code (from the linkage creator), then the main program, then the modules (any order), and then of course the runtime library. The modules can also be assembly language routines. Now when a module needs changing just compile it, assemble it and then link it, this will be much faster than a non-modularized system.

MODULE HEADER FORMAT

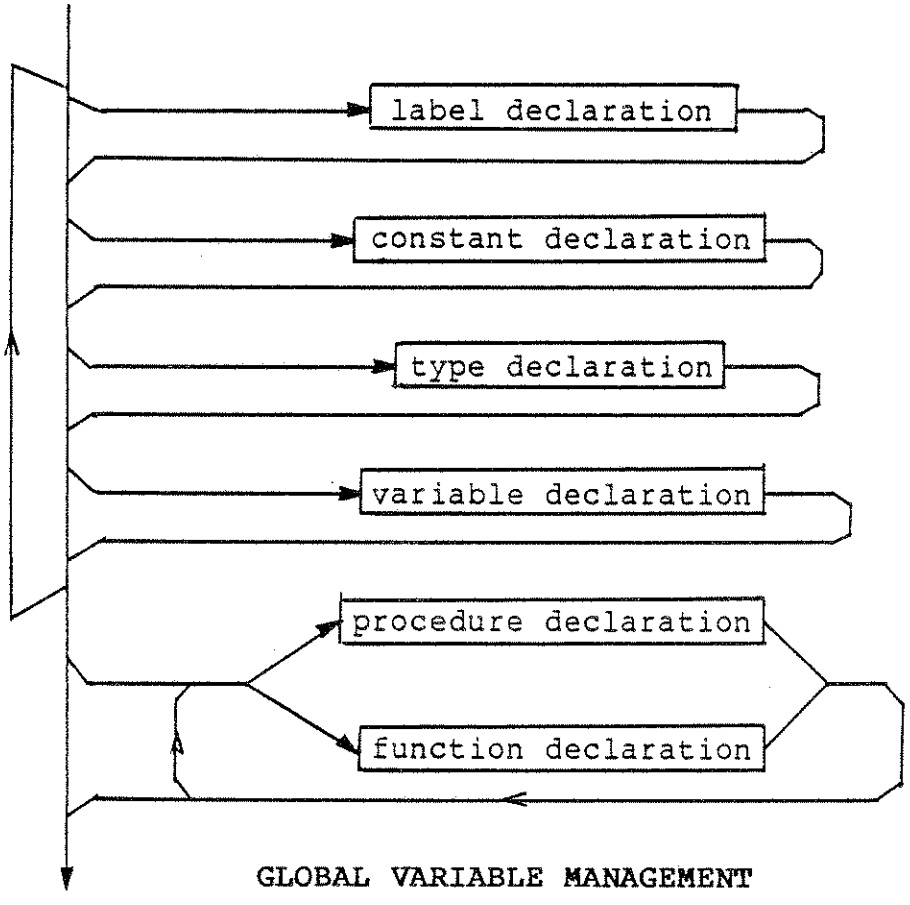
The module header is very similar to a program heading with the word "program" changed to "module". After the last procedure in a module there is the word "modend" and then the period. This replaces the main program's begin end pair.

module heading = **module** identifier [(identifier
{, identifier})] ; sub-block **modend** .



The sub-block is identical to a block except it lacks the begin..statement..end syntax.

```
sub-block = {(label-declaration | constant-declaration |
             type-declaration | variable-declaration)}
             {(procedure-declaration | function-declaration)}
```



GLOBAL VARIABLE MANAGEMENT

If the modules require the use of global variables (including the standard I/O devices) then these must be defined in the modules. Since global variables are assigned stack offsets in the order in which they are compiled, care must be taken to insure that the variable declarations in the modules exactly match the variable declaration in the main program. Failure to observe this precaution will result in execution errors, and possibly the end of life as we now know it.

One of the easiest methods of doing this is to put all of the global variable declarations in an include file and using it in each file. In this manner if a change is made to a global variable it will be changed in each module the next time they are compiled.

If any standard I/O devices are declared in a module they are given the external attribute, therefore they must also be declared in the main program, which will give them the entry attribute.

An example of a system that uses global variables in an include file is the OmegaSoft Screen Editor Kit. This program contains one main program with the main block and the procedures to handle specific terminals. In addition, there are five modules containing somewhat logically related procedures and functions. This scheme makes an otherwise 1500+ line program into small files of around 200 to 400 lines apiece.

EXTERNAL AND ENTRY VARIABLES

Another way to access global variables is to declare variables as entry in one module (normally in the main program) and to declare variables as external in the modules that need to reference them. This method is good for systems that have modules that access only a portion of the global variables. Note that when using this method that any variables declared as entry or external must be unique within the first 6 characters, as they must pass through the assembler and linker.

EXTERNAL AND ENTRY PROCEDURES AND FUNCTIONS

To be used in a modular system some procedures and functions will be declared as entry or external. Only those procedures and functions who must be called from other modules need be declared entry, all others can remain local to their module. Note that any procedures or functions that are declared as entry or external must be unique within the first 6 characters, as they must pass through the assembler and linker.

ASSEMBLY LANGUAGE INTERFACE

In most cases assembly language routines will be called from the Pascal level, this is usually simpler than calling Pascal procedures from assembly language. This is also the more natural direction since Pascal is a high-level language and should handle the outer level of the program with assembly language called to handle low level problems where very fast execution is required.

Although most of this chapter will deal with calling assembly language from Pascal, the details of calling Pascal procedures from assembly language will also be presented.

PARAMETER PASSING

When any procedure or function is called there may be parameters that have to be passed. The parameters are passed on the data (U) stack using PSHU instructions. The parameters are passed in the order that they are declared.

Variable parameters are passed by pushing their address on the data stack.

Value parameters are passed in different ways depending on the data type :

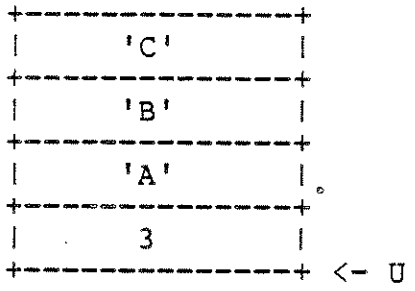
Boolean, character, and enumerated types are simply one byte pushed on the stack.

Integer, hex, and pointer types are two bytes pushed on the stack, most significant byte at the lower address.

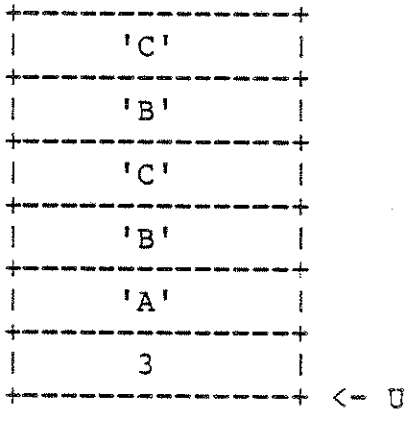
Longinteger, longhex, and real types are four bytes pushed on the stack, most significant (exponent) byte at the lowest address.

Records and arrays are a group of bytes pushed on the stack and appearing just as they do when stored as a variable (the first byte of a record is at the lowest address).

Sets and strings are passed by first pushing the set or string on the stack, with the dynamic length at the lowest address and using dynamic length + 1 bytes on the stack. A procedure is then called which adjusts this variable length structure into a fixed length structure by moving it down to take up as many bytes on the stack as are required by the formal parameter declaration of the called procedure or function. As an example, if the declaration is : string [5] and it is passed by value the string 'ABC' then initially the parameter will occupy 4 bytes on the stack :



After the stack adjustment routine is called the stack will look like :

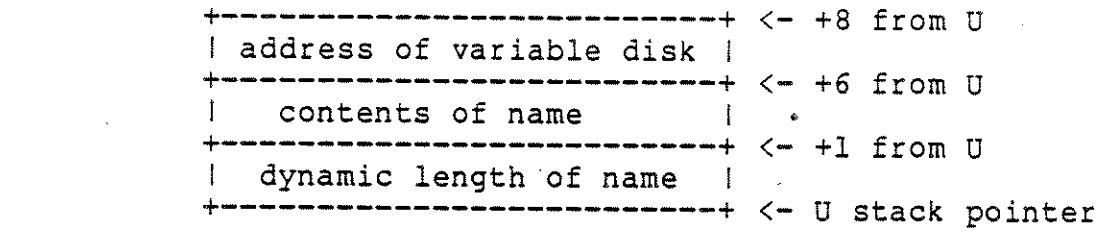


To occupy 6 bytes on the stack. This is done so that for any type of variable and any combinations of parameters a procedure can access its parameters by using fixed offsets from its stack mark.

In the case of assembly language routines there is really no need for all that messing about with stack frames and it is convenient to access parameters referenced off of the U stack mark. Below are two examples of what the stack would look like when control was passed to an assembly language routine :

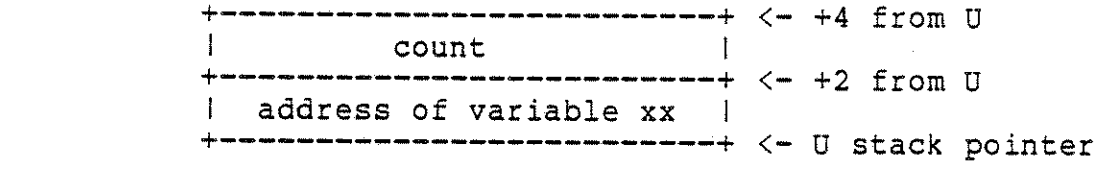
```
procedure test1 (var disk : text ; name : string[5]) ; external ;

function test2 (count : integer ; var xx : string) : boolean ;
                                                      external ;
```

stack frame when testl is called :



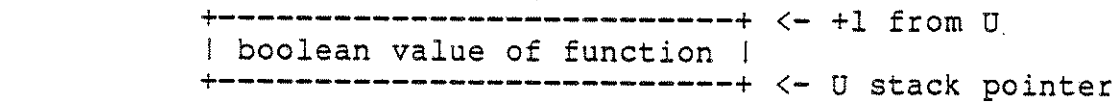

```
to load X with the address of disk : LDX 6,U
to load X with the address of name : LEAX 0,U
to remove parameters from stack    : LEAU 8,U
```

stack frame when test2 is called :




```
to load D with the count           : LDD 2,U
to load x with the address of xx   : LDX 0,U
to remove parameter from stack     : LEAU 4,U
```

When an assembly language procedure returns (RTS) all parameters must be removed by moving the user stack pointer back up. When an assembly language function returns, the only thing on the stack should be the function return value. In the case of test2 above, the stack should look like this when returning :



Function return values are in the same format on the stack as parameters are, except for sets and strings. When a set or string is returned from a function it is not to be adjusted. The set or string must occupy dynamic length + 1 bytes on the stack.

If a Pascal procedure is to be called from assembly language then the assembly language routine must use the same method of parameter passing. In addition to this the Pascal procedure will expect a lexical level difference in the A accumulator. This is the difference between where the caller is and where the called procedure is defined.

For instance if you have an assembly language routine that is at the global level (same as main program) and you call a Pascal procedure that is at lexical level 2 (non-nested procedure) the lexical level difference is :

1 (main program - global) - 2 (procedure) = -1 ($FF)

The base register must also be valid when calling Pascal procedures and in this case the base register (Y register) would be the same as the global base register (see Interrupt procedures).

GLOBAL VARIABLE ACCESSING

Pascal global level variables may be accessed by declaring those variables as entry at the Pascal level and by using an XREF in the assembly language program. Note that for this to work that the base register (Y) at the entry to the assembly language routine must be available (usually best to push it on the system stack if you might need to use the Y register for something else). As an example :

```
flag : byte entry ;
```

may be accessed in assembly language using :

```
 XREF FLAG
   .
   .
 LDX -6,Y GET GLOBAL BASE REGISTER
 LDA FLAG,X GET ADDRESS OF FLAG
```

If the assembly language routine will only be called from the main program block then you can use :

```
 LDA FLAG,Y
```

Since Y will be the global base register.

Two examples will be shown to do the same task, one using Pascal and one using a device driver for access to a memory-mapped CRT. The drivers support the normal carriage return, line feed, and in addition a back space function.

The first example uses the approach of writing the drivers in Pascal and calling the character output routine with one character at a time. The advantages of doing this are that very little assembly language was used (just as an efficient way of initializing the crt controller chip) and the code was developed and debugged very quickly. The disadvantages are not being able to use the standard Write and Writeln statements and the increased size and speed penalty for handling low level I/O in a high-level language.

```
Program crt (Keyboard) ;
{
  Echo characters from the keyboard to the crt. Shown here using
  Pascal to handle the memory-mapped screen. The cursor can be
  moved to anywhere on the screen by setting Xpos and Ypos to
  the desired value.
}
  Const
    Init_size = #15 ;
    Screen_size = 1279 ;
    Xmax = 79 ;
    Ymax = 15 ;
    Cursor = #14 ;
    Space = #$20 ;
    CR = #$D ;
    LF = #$A ;
    BS = #$8 ;

  Type
    MC6845 = Record
               Address : byte ;
               Register : byte
             End ;
    Screen_image = Array [0..Screen_size] of char ;

  Var
    Xpos, Ypos : integer ;
    ascii : char ;
    Crtc : MC6845 at $F804 ;
    Screen : Screen_image at $E000 ;
    initparms : Array [#0..Init_size] of byte pcr ;
```

```
Procedure Init ;
  Var
    Init_count : byte ;
    Screen_count : integer ;
  Begin
    Xpos := 0 ;
    Ypos := 0 ;
    For Init_count := #0 to Init_size do
      begin
        Crtc.address := Init_count ;
        Crtc.register := initparms [Init_count]
      end ;
    For Screen_count := 0 to Screen_size do
      Screen [Screen_count] := space
  End ;

Procedure Crt_out (Character : char) ;
  Var
    Xtemp : integer ;

  Procedure Store (Value : char) ;
    Begin
      Screen [80 * Ypos + Xpos] := Value
    End ;

  Procedure Set_cursor ;
    Begin
      Crtc.address := Cursor ;
      Crtc.register := Chr ((80 * Ypos + Xpos) >> 8) ;
      Crtc.address := Cursor + #1 ;
      Crtc.register := Chr (80 * Ypos + Xpos)
    End ;

  Begin { Crt_out }
    If Character >= Space
      then { not control }
        begin
          Store (Character) ;
          If Xpos = Xmax
            then
              begin
                Crt_out (CR) ;
                Crt_out (LF)
              end
            else
              Xpos := succ (Xpos)
        end
```

REQUIREMENTS FOR O$SET :

On entry X contains the address of the descriptor and A contains the function code.

IF THE FUNCTION CODE = 0 (ST$BRK) :

Then a break check should be done and bit M$BK set or cleared in P$MODE. This is ignored by most drivers. It is used in the driver for INPUT to allow the operator to stop and/or terminate display activity on the CRT or terminate a long execution program.

IF THE FUNCTION CODE = 1 (ST$OPN) :

Then Y points to the file name (terminated with a CR) and B is a flag byte. This entry point is normally used for multi-file devices such as a disk to access one existing file on the device. The status bits of P$MODE have already been set by the Pascal I/O handler. The driver must open the file using the file name in Y. The flags in B are available for system dependent functions. If there are any errors encountered during the open the error code should be set into P$ERR (system dependant).

IF THE FUNCTION CODE = 2 (ST$CRE) :

Then Y points to the file name (terminated with a CR) and B is a flag byte. This entry point is normally used for multi-file devices such as a disk to access one new file on the device. The status bits of P$MODE have already been set by the Pascal I/O handler. The driver must create the file using the file name in Y. The flags in B are available for system dependent functions. If there are any errors encountered during the create the error code should be set into P$ERR (system dependant).

IF THE FUNCTION CODE = 3 (ST$CLS) :

This entry point is normally used for multi-file devices such as a disk to close access to one file on the device. The status bits of P$MODE have already been set by the Pascal I/O handler. The driver must close the file. If there are any errors encountered during the close the error code should be set into P$ERR (system dependant).

IF THE FUNCTION CODE = 4 (ST$DEL) :

Then Y points to the file name (terminated with a CR). This entry point is normally used for multi-file devices such as disk to remove one file from the device directory. That status bits of P$MODE have been set to closed and must remain closed. The driver must delete the file named by Y. If there are any errors encountered during the delete the error code should be set into P$ERR (system dependant).

IF THE FUNCTION CODE = 5 (ST$SEK) :

Then Y points to a 32 bit byte address. This entry point is normally used for random access devices such as disk to position the "window". The MVAL, MEOF, and M$EOLN (if applicable) bits must be cleared after the seek to indicate that the element buffer is not valid. The driver must move the "window" to reflect the byte address pointed to by Y. The driver must not do a data transfer at this point since it does not know the direction of the next transfer. If there are any errors encountered during the seek the error code should be set into P$ERR (system dependant).

IF THE FUNCTION CODE = 6 (ST$PAG) :

Then a Page operation should be done on the Text device. This operation is normally only used for printers and the requirements for pagination vary from printer to printer. The two OmegaSoft standard output devices (OUTPUT and AUXOUT) both support the page operation. There is a byte in the device descriptor to mark whether or not a Writeln was the last operation performed. If it was not one will be generated to make sure the printer is at the beginning of a line. An ASCII form feed will then be sent to the device. One extra byte has been included in the device descriptor that will allow you to keep a line count in cases where the printer does not support form feeds. You could then see how many line feeds were needed to move the printer to the top of the page.

STANDARD DEVICE DRIVERS :

There are five device drivers that are supplied by OmegaSoft as part of the standard runtime package. These are used to handle the standard devices of INPUT, OUTPUT, AUXOUT, and KEYBOARD, The FILE type, and a means for reporting runtime errors. If you wish to re-define the standard devices for your target system, you may modify these drivers in the runtime package.

.INPUT : INPUT
.OUTPU : OUTPUT
.AUXOU : AUXOUT
.KEYBO : KEYBOARD
.DISK : DISK (FILES)
.ERROR : ERROR PRINTER (special format)

FORMAT OF DEVICE DRIVERS :

Each device driver (except .ERROR) has four entry points. These entry points are accessed via the device driver vector table. The table consists of a branch to the appropriate code with the following offsets :

Offset	Label	Destination
0	O$INIT	Initialize device
3	O$XFOT	Transfer data to device
6	O$XFIN	Transfer data from device
9	O$SET	Setup device

If any of the entry points are not used they can be replaced by an RTS. The O$INIT entry will be called when the block is entered that contains the device definition therefore the table must be at the first location for that driver. The U and X registers must not be altered by any of these device drivers.

.ERROR has only one entry point and accumulator A is used to transfer a one byte error code. The OmegaSoft supplied driver will perform a system dependant transformation to the error code, report it in some manner, and return to the operating system. The exception is in the case of error code = 255 in which no error is reported but control is returned to the operating system.

REQUIREMENTS FOR O$INIT :

On entry X contains the address of the descriptor and A contains the mode (only the text, interactive, and break bits are valid, all others are zero). The Y register must not be altered by this entry point. The mode must be stored in P$MODE, P$ERR must be cleared, and the address of the device driver must be stored in P$DRV. If this is a driver for a device that has an implicit open (such as the standard devices) you must also set the appropriate status into P$MODE, set P$ELNT, and setup any extra buffers or flags as required. If the break bit (M$BK) bit is set this indicates that the compiler {$B+} option was on during the initialization of this device. If so, then normally you would clear out this bit in P$MODE and would set a flag of some sort to indicate this option is effective.

REQUIREMENTS FOR O$XFOT :

On entry X contains the address of the descriptor and A contains the writeln flag (text devices only). If A = $01 then a Writeln should be performed on the device. If A = $00 then the data in the element buffer should transferred to the device. If A = $FF then the B register contains a byte count and the U register points one byte below the data to be transferred. The byte count indicates how many bytes starting at (U + 1) are to be transferred to the device. This entry point is used for writing strings to a text device. If any errors occur during the transfer then P$ERR should be set to the appropriate value (System dependant).

REQUIREMENTS FOR O$XFIN :

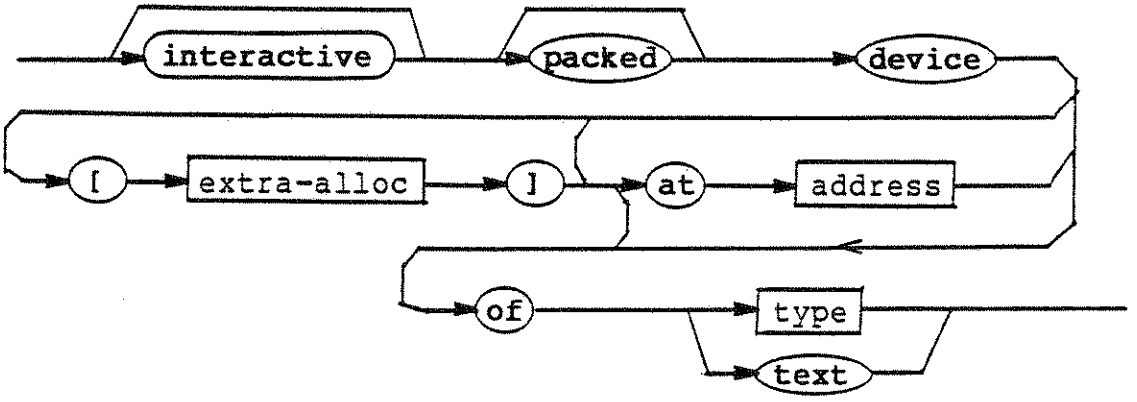
On entry X contains the address of the descriptor. If M$EOF is set in P$MODE then runtime error E$EOF should be set into P$ERR and no transfer should take place. Data should be transferred from the device and stored in the element buffer. If end of file was encountered on that transfer then M$EOF should be set in P$MODE. If this is a text device and the current element is an end of line then M$EOLN should be set in P$MODE or cleared otherwise. If this is a text device and the current element is a ASCII null then a new transfer should take place (ignore nulls). M$VAL should be set in P$MODE to indicate that the contents of the element buffer are valid. If M$EOF or M$EOLN are set in P$MODE of a text device then the element buffer must be set to a space ($20).

WRITING DEVICE DRIVERS

This chapter is meant for those of you are very familiar at assembly language programming. Custom device drivers are useful when you have some I/O device that would fit in well with the Pascal I/O procedures and functions. Examples of possibilities include : printers, plotters, custom disk systems, modems, memory-mapped crts, and networking systems.

When designing your drivers it is recommended that you look over the drivers that came with your system for the standard devices and make sure you understand what they do and how they do it.

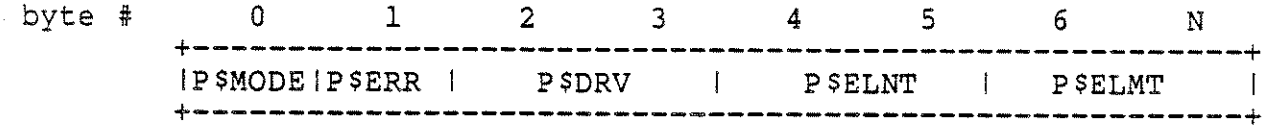
```
device-type = [interactive] [packed] device [ [ extra-alloc ] ]
              [ at address ] of (type | text)
extra-alloc = (unsigned-integer-constant | hex-constant)
address = (unsigned-integer-constant | hex-constant)
```



Devices are the entity that represent I/O devices in a system. As is true with all types, PACKED has no meaning in this compiler and is ignored. If INTERACTIVE is specified it sets the M$INT bit in the mode byte which will affect the meaning of the EOF and EOLN flags. If the integer/hex constant in brackets is specified it is the amount of extra stack space to be allocated for the device descriptor in addition to the minimum amount required. This is normally used to reserve space for flags or buffers that are normally device and/or system dependant. If the integer/hex constant is specified following the word "at" it is the address of the device driver vector table. If this value is not specified the device type name is used preceded by a period as the address of the table.

On the following pages is the format of the device descriptor and the function of each entry point in the device driver.

FORMAT OF A DEVICE DESCRIPTOR :

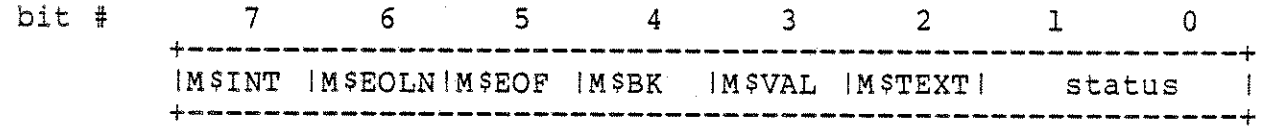


where :

P$MODE is the current mode/flags
P$ERR is the error status of the device
P$DRV is the address of the device driver vector table
P$ELNT is the length (in bytes) of the element
P$ELMT is the element

There may be additional bytes after the element to hold flags or buffers.

FORMAT OF THE MODE BYTE :



where :

M$INT device is interactive if set
M$EOLN current element is end of line if set
M$EOF current element is end of file if set
M$BK break occurred on device if set
M$VAL current element is valid if set
M$TEXT device is text device if set

status has four possible states

0 - M$CLOS device is closed
1 - M$IN device is open for input
2 - M$OUT device is open for output
3 - M$UPDT device is open for input and output

INTERRUPT PROCEDURES

Interrupts are not explicitly handled by OmegaSoft Pascal. The interrupt mask is not modified by the Pascal or by any of the runtime routines. We considered being able to define at the Pascal level 'Interrupt Procedures' but due to the many ways interrupts can be handled (You have NMI, IRQ, FIRQ, possibility of one device per interrupt, priority interrupt controllers, interrupt polling, many different type of operating system interrupt restrictions, etc.) it was decided to let you do it yourself!

The procedure used to handle an interrupt (assuming it is not all done in assembly language) must be given the entry attribute and be at lexical level 2. When an interrupt occurs the base register must be loaded with the global stack mark and accumulator A is loaded with -1 (you are calling lexical level 2 from lexical level 1 (global)) and the procedure is called. This is followed by a RTI assembly language instruction. As an example the following routine will read a port to clear the interrupt flag and update a timer to be used as a delay mechanism in the main Pascal program.

```
procedure rtcirq ; entry ;
  var
    dummy : byte ;
  begin
    dummy := port { read port to clear interrupt }
    if time <> 0
      then
        time := pred (time) { update timer if not zero }
  end ;
```

Assembly language support :

```
IRQVEC LDY #global stack mark
       LDA #-1
       LBSR RTCIRQ
       RTI
```

```
      else { control character }
        Case Character of
          CR : Xpos := 0 ;
          LF : begin
                 Ypos := succ (Ypos) ;
                 If Ypos > Ymax
                   then
                     Ypos := 0 ;
                 Xtemp := Xpos ;
                 For Xpos := 0 to Xmax do
                   Store (space) ;
                 Xpos := Xtemp
          BS : If Xpos <> 0
                 then
                   begin
                     Xpos := pred (Xpos) ;
                     Store (space)
                   end
          end ;
    Set_cursor
  End ;

Begin { CRT }
  Init ;
  Repeat
    Read (Keyboard, ascii) ;
    Crt_out (ascii)
  Until false
End .
```

The following is a table used to initialize the crt controller chip. Note how this table was accessed by defining it as pcr in the Pascal program.

```
        NAM CRTINT
        TTL INIT TABLE FOR CRT CONTROLLER
*
        XDEF INITPA
*
* INIT PARAMETERS FOR MC6845
*
INITPA EQU *
        FCB $62 HORIZONTAL TOTAL
        FCB $50 HORIZONTAL DISPLAYED
        FCB $54 H. SYNC POSITION
        FCB $02 H. SYNC WIDTH
        FCB $14 VERTICAL TOTAL
        FCB $7 V. TOTAL ADJUST
        FCB $10 VERTICAL DISPLAYED
        FCB $12 V. SYNC POSITION
        FCB $0 INTERLACE MODE
        FCB $B MAX SCAN LINE ADDRESS
        FCB $B CURSOR START
        FCB $B CURSOR END
        FDB $0 START ADDRESS
        FDB $0 CURSOR ADDRESS
        END
```

To link these two files together with the runtime library one would have the following load command for the linker :

```
LOAD=CRT.CA CRT.PA CRTINT
```

The second example uses the approach of writing the drivers in assembly language as an actual device driver. The advantages of doing this are the ability to use the standard Write and Writeln statements and reduced program size and higher transfer speed. The disadvantage is of programming in assembly language thereby increasing programming and debugging time.

```
Program video (Keyboard) ;
{
  Echo characters from the keyboard to the crt. Shown here using
  A device driver to handle the memory-mapped screen. The cursor
  can be moved to anywhere on the screen by using a :
  Seek (Crt, <256*Ypos + Xpos>).
}
  Var
    ascii : char ;
    crt : Device [2] of char ;
  Begin
    Repeat
      Read (Keyboard, ascii) ;
      Write (crt, ascii)
    Until false
  End.
```

The following is the actual device driver code that would be linked with the above Pascal program.

```
        NAM VIDEO1
        TTL DEVICE DRIVER FOR CRT
*
        INCL PASEQU
*
P$XPOS  EQU 7 THESE ARE THE TWO EXTRA BYTES RESERVED
P$YPOS  EQU 8 BY THE DECLARATION DEVICE [2] OF TEXT ;
BS      EQU $8

        XDEF .CRT
*
* DRIVER VECTOR TABLE HAS 4 ENTRIES
*
* + 0 - INITIALIZATION
* + 3 - OUTPUT
* + 6 - INPUT
* + 9 - SETUP
*
.CRT    BRA CRINIT
        RMB 1
        BRA CROUT
        RMB 1
        RTS
        RMB 2
        LBRA CRSET
```

```
*
* INIT CRT
*
CRINIT PSHS Y
       ORA #M$OUT OPEN FOR OUTPUT DURING INIT
       STA P$MODE,X
       CLR P$ERR,X
       LEAY .CRT,PCR ADDRESS OF VECTOR TABLE
       STY P$DRV,X
       LDY #1
       STY P$ELNT,X BASIC ELEMENT IS CHARACTER (1 BYTE)
       CLR P$XPOS,X
       CLR P$YPOS,X SET TO LEFT TOP OF SCREEN
       LEAY INIT,PCR VALUES FOR MC6845
       CLRB
CRIN1  STB $F804
       LDA B,Y
       STA $F805
       INCB
       CMPB #16
       BLO CRIN1 SETUP FIRST 16 REGISTERS
       LDA #SPACE
       LDY #$E000
CRIN2  STA 0,Y+ SPACE OUT SCREEN MEMORY
       CMPY #$E000+1280
       BLO CRIN2
       PULS Y,PC
*
* OUTPUT CHARACTER IN ELEMENT BUFFER
*
CROUT  LDA P$ELMT,X GET IN ACCUM FOR REST OF ROUTINE
CROT1  CMPA #SPACE
       BLO CTL1 HANDLE CONTROL CHARACTERS
       LBSR STORE PUT IN SCREEN MEMORY
       LDA P$XPOS,X
       CMPA #79 SEE IF ON LAST COLUMN
       BNE NCTL1 NOPE
       LDA #CR
       BSR CROT1 BRING BACK TO START OF LINE
       LDA #LF
       BSR CROT1 AND DOWN TO NEW LINE
       BRA CROT2 FOR AUTO CR/LF AT END OF LINE
NCTL1  INC P$XPOS,X NEXT CHARACTER POSITION
       BRA CROT2
CTL1   CMPA #CR
       BNE CTL2
       CLR P$XPOS,X MOVE TO FRONT OF LINE
       BRA CROT2
```

```
CTL2    CMPA #LF
        BNE CTL3
        INC P$YPOS,X MOVE TO NEXT LINE
        LDA P$YPOS,X
        CMPA #16 SEE IF GONE TOO FAR
        BLO CTL4 NOPE
        CLR P$YPOS,X START BACK AT TOP OF SCREEN
CTL4    LDA P$XPOS,X SAVE CURRENT X POSITION
        PSHS A
        LDB #80
        CLR P$XPOS,X
CTL5    LDA #SPACE
        PSHS B
        BSR STORE CLEAR OUT NEW LINE WITH SPACES
        PULS B
        INC P$XPOS,X NEXT CHARACTER
        DECB FOR FULL 80 COLUMNS
        BNE CTL5
        PULS A
        STA P$XPOS,X RESTORE X POSITION
        BRA CROT2
CTL3    CMPA #BS
        BNE CROT2
        TST P$XPOS,X SEE IF CAN BACKUP
        BEQ CROT2 NOPE, AT START ALREADY
        DEC P$XPOS,X
        LDA #SPACE
        BSR STORE AND WIPE OUT OLD CHARACTER
CROT2   BSR CALC GET BUFFER OFFSET IN ACC. D
        PSHS B
        LDB #14
        STB $F804
        STA $F805 MS BYTE OF CURSOR ADDRESS
        INCB
        STB $F804
        PULS A
        STA $F805 LS BYTE OF CURSOR ADDRESS
        RTS
*
* PUT CHARACTER IN ACC. A INTO SCREEN MEMORY
*
STORE   PSHS X,A
        BSR CALC
        ADDD #$E000 ADD TO START OF MEMORY
        TFR D,X
        PULS A
        STA 0,X
        PULS X,PC
```

```
*
* CALCULATE OFFSET INTO SCREEN MEMORY, RETURN IN D
*
CALC    LDA P$YPOS,X
        LDB #80
        MUL
        ADDB P$XPOS,X
        ADCA #0
        RTS
*
* HANDLE SEEK - SET X AND Y TO CORRESPOND TO RELATIVE
* SCREEN POSITION
*
CRSET   CMPA #ST$SEK SEE IF REALLY SEEK
        BNE CREX NOPE, IGNORE ANY OTHER REQUEST
        LDD 2,Y GET LS 16 BITS OF 32 BIT NUMBER
        STA P$YPOS,X MS IS Y POSITION
        STB P$XPOS,X LS IS X POSITION
CREX    RTS
*
* INIT PARAMETERS FOR MC6845
*
INIT    EQU *
        FCB $62 HORIZONTAL TOTAL
        FCB $50 HORIZONTAL DISPLAYED
        FCB $54 H. SYNC POSITION
        FCB $02 H. SYNC WIDTH
        FCB $14 VERTICAL TOTAL
        FCB $7 V. TOTAL ADJUST
        FCB $10 VERTICAL DISPLAYED
        FCB $12 V. SYNC POSITION
        FCB $0 INTERLACE MODE
        FCB $B MAX SCAN LINE ADDRESS
        FCB $B CURSOR START
        FCB $B CURSOR END
        FDB $0 START ADDRESS
        FDB $0 CURSOR ADDRESS
        END
```

To link these two files together with the runtime library one would have the following load command for the linker :

```
LOAD=VIDEO.CA VIDEO.PA VIDEO1
```

Another good source of information on device drivers is to look at the standard device drivers that came with the runtime package. These will be in files SD1-SDx.

INDEX

A

B

C

D

D

E

F

G

H

I

L

M

N

O

P

R

S

S

T

U

V

W

APPENDIX

APPENDIX A - COMPILATION ERRORS

1 Invalid token
  * A character was encountered in the source that is not a valid pascal character.
2 Invalid char
  * Something other than a valid integer or hex number followed the "#" character.
3 Invalid integer
4 Invalid hex
  * Value exceeded $FFFF.
5 Invalid longhex
  * Value exceeded $FFFFFFFF.
6 Invalid longinteger
7 Invalid real
  * Real number too large.
8 . expected
  * After the end of the global block or after the modend of a module there should be a period.
9 Identifier expected
  * The syntax calls for an identifier, either defined or undefined.
  * In a record variant declaration an identifier must be present as the tag type or variable.
  * In a variable declaration or procedure parameter list at least on identifier must be provided before the colon.
10 (expected
  * After the program or module name there should either be a left paren or a semicolon.
  * In a record variant declaration a left paren is expected after the colon the follows the case constant.
  * A left paren is expected to start the parameter list required.
11) expected
  * In a record variant declaration a right paren is expected after a field list.
  * In an enumerated type declaration either a comma or a right paren was expected to close the enumeration.
  * In a procedure parameter list a right paren must close the parameter list.
  * A right paren was not found after the expression of a nested factor.
  * A right paren was not found after the parameter of an exit statement.
  * A right paren was not found after the parameter list for a standard procedure or function.
12 Error in parameter list
  * The program parameter list was not correctly terminated by a right paren.

13 ; expected
* After the program parameters a semicolon should appear.
* In the label declaration syntax either a comma or semicolon should appear here.
* In the constant declaration syntax a semicolon should be found after each constant value.
* In the type declaration syntax a semicolon should be found after the type definition.
* In the variable declaration syntax as semicolon should be found after the type definition.
* In a procedure declaration a semicolon should be found after the parameter list (if provided) or else after the procedure name.
* After the block of a procedure should appear a semicolon.
* A semicolon should separate statements in a compound or repeat statement.

14 Invalid string
* string crosses a line boundary.

15 Identifier declared twice
* In a definition an identifier was used that has already been defined and cannot be re-defined at this lex level.
* In a record variant declaration the tag variable identifier is already defined in the record.
* In an enumerated type declaration a name was repeated.
* In a procedure declaration the procedure name was already defined but was not forward.

16 = expected
* In the constant declaration syntax an equal should occur after the identifier.
* In the type declaration syntax an equal should occur after the identifier.
* An equal sign was expected in the expression rather than the assign symbol found.

17 Invalid constant
* In the constant syntax a valid expression was found but it was not constant.

18 Undefined identifier
* The identifier referenced was not found in the symbol table.
* At the start of statement an integer was found but was not declared as a label.
* The destination integer for a GOTO was not defined as a label.
* The identifier as not valid as the start of a factor.

19 Identifier not appropriate class
* In a record variant declaration the token found for the tag field type was not a type.
* A record selector was found that is not part of the record.
* The token should of been a variable but was not.
* The argument of a with statement must be a record.
* The parameter for the sizeof function must be a variable or type.

20 END or ; expected
* In a compound statement either an end to terminate it or a semicolon to extend it was not found.
* In a record declaration either a new field list or end of record was expected.
* In a case statement either a new case constant list, an else/otherwise clause, or an end was expected.

21 .. expected
* In a simple type declaration the ".." was not found after the first constant.

22 Error in simple type
* Where a simple type was expected something else was found.

23 Incompatible subrange types
* In a simple type declaration the type of the second constant did not match that of the first constant of the subrange.

24 Low bound exceeds high bound
* In a simple type subrange declaration the first constant had a higher value than the second constant.

25 Invalid subrange type
* In a simple type declaration a constant of a type that cannot have a subrange was found.
* The two expression types found in a subrange set constructor do not match.
* The constant type is not valid for a subrange case constant.

26 OF expected
* OF expected in record variant declaration.
* In a set declaration OF was expected.
* In a device declaration OF was expected.
* In an array declaration OF was expected.
* An OF is expected after the selector in a case statement.

27 [expected
* In an array declaration a left paren was expected to start an indices declaration.

28] expected
* In a device declaration a right paren was expected after the extra stack space specification.
* In an array declaration a right paren was expected after the indices declaration.
* In a string declaration a right paren was expected after the size specification.
* In array indexing a right bracket was not found to terminate the indexing.
* A right bracket was not found after the expression in string indexing.
* A right bracket was expected to close a set constructor.

29 Error in type

30 Integer expected
* In a device declaration either an integer or hex number was expected for a extra stack space specification.
* In a device declaration either an integer or hex number was expected for the absolute init address.
* In a string declaration an integer was expected as a size specification.
* The token following a GOTO must be an integer.
* The expression for string indexing must be an integer.
* The parameter of an exit statement must be an integer.

31 : expected
* In a record variant declaration a colon is expected after the case constant.
* In a variable declaration or a procedure parameter list a colon is expected after the variable list and before the type definition.
* In a function declaration a colon is required after the parameter list and before the function return type definition.
* A colon must appear after the constant list in a case statement.

32 Stack allocation overflow
* While allocating stack space for a variable the allocation counter overflowed.

33 Wrong index type
* The array index expression type does not match the declaration.

34 Invalid index type
* The type declared as an array indices type cannot be used.

35 Error in variable

36 := expected
* An assignment symbol should appear after the variable specification in an assignment or for statement.

37 Type conflict
* The expression found does not have an acceptable type.
* The expression type is not compatible with the variable type of an assignment or for statement.
* The two sides of an expression do not have matching types.
* The constant type does not match the selector type in a case statement.
* The parameter for the mark or release procedure must be integer or hex.
* The parameter for a read or write procedure to a non-text device does not match the device declaration.
* The parameter of the random function must be a real.

38 Value is out of bounds

39 Invalid type of operand
* The not operation was applied to an incorrect type.
* An incorrect type was used for set constuction.
* The operand(s) type cannot be used with the specified operator.

40 Address must be integer or hex
* In a variable or procedure declaration the constant following the "at" was not integer or hex.

41 Zero string not allowed
* In a string declaration a size of zero was specified.

42 Operator expected

43 , expected
* In array indexing neither a comma or a right bracket were found after the index expression.
* In a parameter list more elements are in the declaration and a comma is expected to separate the next expression.

44 Error in type of standard procedure parameter
* This is general catch-all for errors in the parameter lists of standard procedures. Re-read the documentation for the procedure where this error occurs.

45 Undeclared or incorrect standard I/O device
* A default device was used for a standard procedure or function and the default device was not declared in the program parameter list.

46 Error in type of standard function parameter
* This is general catch-all for errors in the parameter lists of standard functions. Re-read the documentation for the function where this error occurs.

47 Invalid type of expression
* A boolean expression must be used for the while, repeat, and if statements.
* The expression type is invalid for a for statement.
* The expression type is invalid for a case statement selector.

48 DO expected
* DO should be after the expression of a while statement.
* DO should be after the second expression of a for statement.
* DO should be after the argument list of a with statement.

49 THEN expected
* After the expression of an IF statement a THEN should be found.

50 Number of parameters incorrect
* A user defined procedure was called with the incorrect number of parameters.

51 Invalid parameter substitution
* A user defined procedure call encountered a parameter that does not match the definition.

52 Backtrack error
* Error in symbol table management - contact OmegaSoft if you ever get this error.

53 File value parameter not allowed

54 TO/DOWNTO expected
* A TO or DOWNTO is expected after the first expression in a for statement.

55 Max string length is 126
* A string was defined with a length in excess of 126 characters. The length was set to 126.

56 Too many set elements
* The maximum ordinal value for a base element is 1007.

57 Invalid base type
* A pointer was declared to an invalid base type.
* A set was declared with an invalid base type.

58 FATAL ERROR - SYSTEM STACK OVERFLOW
* Expressions or statements are nested too deep.

59 FATAL ERROR - SYMBOL TABLE OVERFLOW
* Too many user defined identifiers or excessive expression or statement nesting.

60 PROGRAM expected
* The words program or module must be the first non-comment token in the file.

61 Case constant type conflict
* In a record variant declaration the case constant does not match the selector type.

62 Exit outside of loop
* Not nested deep enough in loops to exit the number of loops specified.

63 Forward pointer reference not defined
* At the end of a type declaration, variable declaration, or procedure parameter declaration there was a reference to but no declaration for the named identifier as a pointer base type.

64 Compiler out of sync or no "."
* End of file was encountered before the compiler expected it. Either the period was left off of the end of the program or the compiler has gotten out of sync due to a syntax error and not been able to restore itself to proper parsing.

65 Wrong lexical level
* In a variable declaration either absolute, external, entry, or pcr addressing was specified at a non-global level.
* In a procedure declaration either absolute, external, or entry addressing was specified at a non-global level.

66 Invalid GOTO to label
* Cannot jump into a syntax that has temporary values on the data stack (such as a for loop).

67 Constant out of range
* The value to be assigned to a subrange variable is out of range.
* The index expression for a variable was beyond the range of the declaration.

68 FATAL ERROR - VARIABLE CODE BUFFER OVERFLOW
* Code in excess of what the buffer would allow was generated to access the variable of an assignment statement.

69 Forward buffer overflow
* Too many pointer type declarations in this declaration section.
* Too many variables or procedure parameters defined to be pointer types in this declaration.
* Too many GOTO statements with destination declared after the GOTO in this block.

70 Variable buffer overflow
* In a variable declaration or a procedure parameter list too many variable names were listed before the type definition.
* Too many arguments for or nested with statements.

71 Program parameter buffer overflow
* There are too many identifiers in the program parameter declaration for the compiler to handle.

72 Statement expected
* Where a statement was expected the token found does not represent a valid start of statement.

73 Invalid function return type
* A function return type was specified that has a size of over 127 bytes.

74 BEGIN expected
* The block syntax expected a begin to start a compound statement.

APPENDIX B - RUNTIME ERRORS

There are four types of runtime errors : range, I/O, conversion, and stack. The first three are individually maskable using the R, I, and C controls in the comment syntax. Stack errors are not maskable since they represent a limit exceeded and that the necessary information on the stack may no longer be valid enough to continue execution. Error codes 4 through 22 are defined for all systems and are listed here.

RANGE ERRORS

Number	Error	Where generated
4	Overflow	Integer add, subtract, mul. and negate. Hex mult. Real add, subtract, mult, divide. Log functions, Arcsin/cos
5	Truncation	Chr func. Character mult. Real to integer conver.
6	Divide by zero	All div, mod operations and real /
7	Invalid argument for square root or log	Square root, logs
8	Dynamic length assign error	String/set assignment size adjust (String/set value parameters) String concat.
9	Array index invalid	Array indexing
10	String index invalid	String indexing, substr
11	Subrange error	assignment to subrange var.
12	Set element out of range	set construction

CONVERSION ERRORS

Number	Error	Where generated
13	Integer conversion err.	Read of integer or longinteger, integer or longinteger function with string parameter.
14	Hex conversion error	Read of hex or longhex, hex or longhex function - string param
15	Real conversion error	Read of real, real function
16	Boolean conversion err	Read of boolean, boolean func.

I/O ERRORS

Number	Error	Where generated
17	Device open/closed incorrect	Get, Put, Read, Readln, Page Write, Writeln, Open, Create Seek, Del
18	Device hit EOF	Get, Read, Readln

STACK ERRORS

Number	Error	Where generated
19	Heap overflow	Heap crashed into data stack or heap limit during NEW
20	Data stack overflow	Data stack crashed into heap or data stack limit during stack allocation
21	System stack overflow	System stack crashed into system stack limit during Proc/Func calling

USER GENERATED ERRORS

Number	Error	Where generated
22	Program operation error	Halt(22) or Halt(#22)

APPENDIX C - RUNTIME ENVIRONMENT

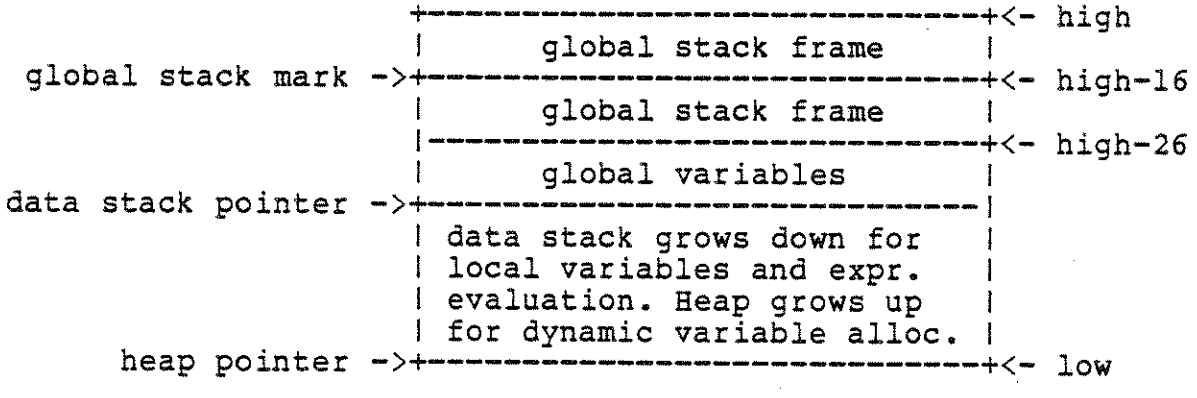
The are four memory areas used during the execution of a Pascal program, two of which may be the same area.

The first is the actual program code, this may be in Ram or Rom (or any other type of memory that is read only, or read-writeable).

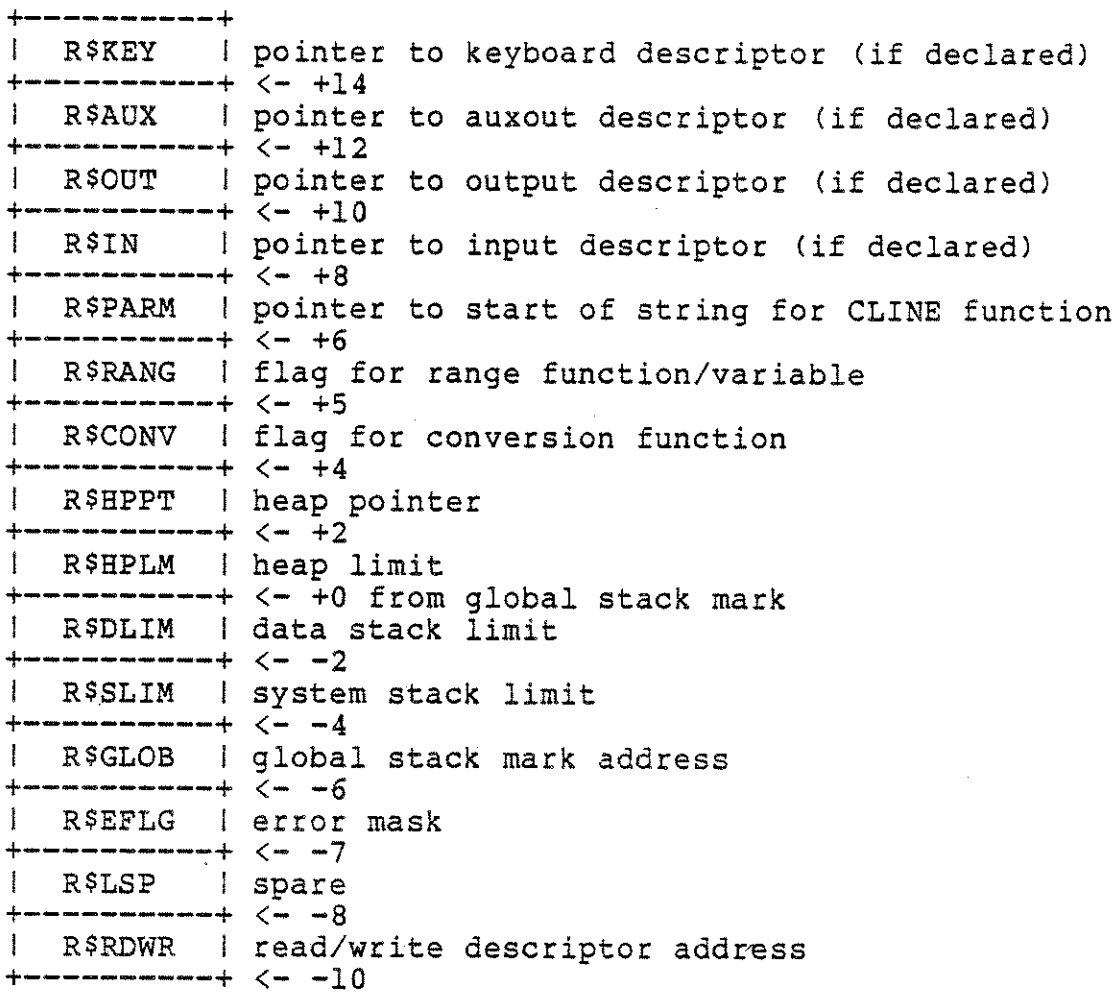
The second is the system stack which is used for calling procedures, functions, and the assembly language routines in the runtime library. Temporary values may also be stored on the system stack by the runtime routines. Interrupts will also extend the stack if used, extra room should be allocated for those if used. Some of the 6809 operating systems use software Interrupts for system calls, these can increase the system stack requirements drastically.

The third and fourth areas are for the data stack and the heap. These can be separated into two areas (as long as the heap is below the data stack) but are more commonly shared in one block of memory. In this case the heap starts at the lower limit of this memory area and builds up when the NEW procedure is called. The data stack starts at the high limit of this memory and builds down as variable space is allocated and as expressions are evaluated. An error will occur if the data stack collides into heap pointer or if the heap pointer collides into the data stack.

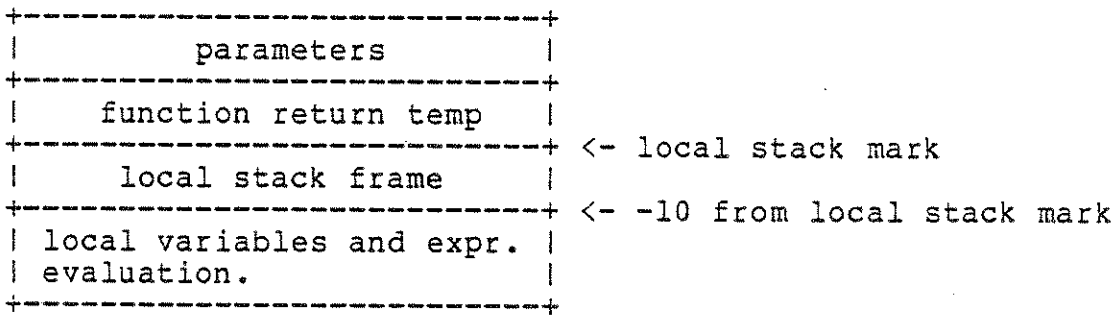
We now present the heap/data stack areas graphically :



global stack frame :



Procedure/Function stacking :



local stack frame :

```
+-----------+ <- -0 from local stack mark
|  R$SLNK   | static link
+-----------+ <- -2
|  R$DLNK   | dynamic link
+-----------+ <- -4
|  R$GLOB   | global stack mark address
+-----------+ <- -6
|  R$EFLG   | error mask
+-----------+ <- -7
|  R$LSP    | spare
+-----------+ <- -8
|  R$RDWR   | read/write descriptor address
+-----------+ <- -10
```

format of error mask

```
bits   7       .....       3      2        1        0
     +-------------------------+--------+--------+--------+
     |        not    used      | R$ECON | R$EIOE | R$ERNG |
     +-------------------------+--------+--------+--------+
```

Range errors (4-12) disabled if R$ERNG zero (R$RANG is set if range error regardless of the state of R$ERNG).
I/O errors (17,18) disabled if R$EI0E zero.
Conversion errors (13-16) disabled if R$ECON zero (R$CONV is updated regardless of the state of R$ECON).

The static link is the address of the stack mark of the level enclosing this one (lexical) so that variables can be accessed properly at other levels. The dynamic link is the address of the stack mark of the caller. This is restored as the stack mark when this called procedure/function returns.

The CPU registers are used as follows during runtime :

PC - program counter - program execution address
S - system stack pointer
U - data stack pointer
Y - stack mark (base register)
X - general purpose address and data register
D - general purpose data register

APPENDIX D - RUNTIME ROUTINES

Entry point to file cross reference :

entry	file	entry	file	entry	file	entry	file
FNC00	SO1	FNC01	SF15	FNC02	SF22	FNC03	SF22
FNC04	SF20	FNC09	SF18	FNC10	SF16	FNC11	SF16
FNC12	SF16	FNC13	SP13	FNC20	SF1	FNC21	SF1
FNC22	SF1	FNC23	SF21	FNC24	SF21	FNC25	SF21
FNC26	SF20	FNC30	SO3	FNC31	SF4	FNC32	SF4
FNC33	SF5	FNC34	SF6	FNC35	SF7	FNC36	SF7
FNC37	SF8	FNC38	SF7	FNC39	SF9	FNC40	SF9
FNC41	SF14	FNC42	SF15	FNC50	SF10	FNC51	SF10
FNC53	SF10	FNC60	SF11	FNC61	SF11	FNC62	SF12
FNC63	SF12	FNC64	SF13	FNC65	SF13	FNC66	SF19
FNC67	SF19	FNC68	SF23	FNC69	SF23	FNC70	SF17
FNC71	SF24	FNC72	SF24	FNC73	SF26	FNC74	SF26
INT00	SU4	INT01	SU7	INT02	SU5	INT03	SU6
INT04	SO6	INT05	SU2	INT06	SF1	INT07	SF1
INT09	SO3	INT10	SU2	INT11	SO3	INT12	SO3
INT14	SP7	INT16	SF2	INT17	SF2	INT18	SF3
INT20	SU8	INT21	SU2	INT22	SU2	INT23	SU8
INT24	SU3	INT25	SU3	INT26	SU8	INT27	SU3
INT28	SU3	INT29	SU3	INT30	SU9	INT31	SU9
INT32	SU12	INT33	SU10	INT34	SO6	INT35	SU11
INT36	SU11	INT37	SU12	INT38	SU7	INT40	SF20
INT41	SF20	INT42	SF21	INT43	SF21	INT44	SF25
INT45	SF25	INTF8	SU1	INTF9	SU1	INTFA	SU1
INTFB	SU1	INTFC	SU1	INTFD	SU1	INTFE	SU1
INTFF	SU1	OPR02	SO1	OPR03	SO1	OPR04	SO1
OPR05	SO11	OPR06	SO11	OPR07	SO12	OPR08	SO12
OPR09	SO12	OPR10	SO11	OPR11	SO11	OPR12	SO2
OPR13	SO2	OPR14	SO2	OPR16	SO8	OPR17	SO8
OPR20	SO3	OPR21	SO3	OPR22	SO3	OPR23	SO3
OPR24	SF7	OPR25	SO4	OPR26	SO3	OPR30	SO6
OPR31	SO6	OPR32	SO7	OPR33	SO7	OPR35	SO6
OPR36	SO10	OPR37	SO10	OPR38	SO10	OPR39	SO10
OPR40	SO9	OPR41	SO13	OPR42	SO13	OPR43	SO13
OPR44	SO14	OPR45	SO14	OPR46	SO15	OPR47	SO15
OPR48	SO15	OPR49	SO15	OPR50	SO5	OPR51	SO16
OPR52	SO16	OPR53	SO16	PRC00	SP10	PRC01	SP10
PRC02	SP11	PRC03	SP11	PRC04	SP12	PRC05	SP12
PRC06	SP8	PRC07	SP8	PRC08	SP9	PRC09	SP9
PRC10	SP6	PRC11	SP6	PRC12	SP7	PRC13	SP14
PRC15	SP1	PRC16	SP1	PRC18	SP3	PRC19	SP3
PRC20	SP3	PRC21	SP4	PRC22	SP5	PRC23	SP2
PRC25	SP14	PRC26	SP14	PRC31	SP2	PRC32	SP2
PRC34	SP2	PRC35	SP2	PRC36	SP13	PRC37	SP13
PRC38	SP13	PRC39	SP13	PRC40	SP10	PRC42	SP11
PRC44	SP12	PRC46	SP8	PRC48	SP9	PRC50	SP6
PRC52	SP15	PRC54	SP15	PRC55	SP15	PRC60	SP17
PRC61	SP16	PRC62	SP16	PRC63	SP18	PRC64	SP18
PRC65	SP18						

APPENDIX E - ISO VALIDATION REPORT

Pascal Processor Identification

Host Computer : Smoke Signal Broadcasting Chieftain 9522812W10 running the OS-9 operating system.

Host Computer Requirements : MC6809 processor, minimum of 56k bytes of memory, 2 or more disk drives, running the OS-9, MDOS, XDOS, or FLEX operating system.

Processor : OmegaSoft Pascal version 2.10

Test Conditions

Tester : R.D. Reimiller

Date : June 1982

Validation Suite Version : 3.0

General Introduction to the OmegaSoft Implementation

The OmegaSoft Pascal compiler was developed to provide the users of the 6809 processor with a fast and efficient way to develop code capable of running on the host development system or installed into a target system. The compiler is aimed primarily at industrial applications such as process control and instrumentation. Due to the nature of these applications many extensions were added such as byte arithmetic, long integers, dynamic length strings, modular compilation, and versatile variable addressing. As a secondary requirement it was desired that the compiler be able to accept a Pascal program written in ISO standard Pascal wherever possible.

Conformance Tests

Number of tests passed = 145
Number of tests failed = 11 (8 reasons)

Details of Failed Tests

Test 6.4.2.3-3 : If an enumerated type is defined in the index declaration part of an array its values cannot be referenced until the array declaration is complete.

Test 6.4.2.3-4 : If an enumerated type is defined in a record its values cannot be referenced until the record declaration is complete.

Tests 6.6.3.1-4, 6.6.3.4-1, 6.6.3.4-2, and 6.6.3.5-1 : Procedures and functions cannot be passed as parameters.

Test 6.6.5.4-1 : Pack and Unpack procedures are not supported.

Test 6.7.2.2-3 : Failed on MOD using a negative dividend. The Jenson/Wirth "remainder after division" method is used rather than the method specified in the ISO standard.

Test 6.8.2.4-1 : Non-local GOTO's are not allowed.

Test 6.8.3.9-1 : Assignment to the control variable of a FOR loop occurs after the evaluation of the first expression.

Test 6.9.3.5.1-1 : Real numbers written out in floating point format always have six digits to the right of the decimal point.

Deviance Tests

Number of deviations correctly detected = 83
Number of tests showing true extensions = 45 (22 reasons)
Number of tests not detecting erroneous deviations = 9 (6 reasons)

Details of Extensions

Test 6.1.5-4 : No digits are needed after the decimal point in a real number.

Tests 6.1.6-4 and 6.1.6-5 : Labels may be an positive integer constant.

Tests 6.1.7-5, 6.4.3.1-3, 6.4.3.1-4, 6.6.3.3-5, 6.9.3.2-2 : All variables are packed at the byte level, the reserved word "Packed" is ignored in any type declaration.

Tests 6.1.7-6, 6.1.7-7, 6.1.7-8, 6.4.3.2-5 : Strings, characters, and arrays of less than 127 elements are all compatible.

Tests 6.1.7-11 and 6.4.5-12 : Strings are dynamic length, allowable length is from 0 (null string) to 126.

Tests 6.2.1-8 and 6.2.1-10 : Label, const, type, and var declaration sections can be in any order and repeated multiple times until a procedure/function declaration or "begin" is encountered.

Test 6.3-9 : In any context where a constant is acceptable an expression with a constant value may be used.

Test 6.4.2.3-5 : All enumerated type values are compatible.

Test 6.4.3.3-8 : The values of the case constants in a record variant declaration are not used, access is provided to all variants at all times.

Test 6.4.5-7 : All subranges of the same type are compatible.

Test 6.4.5-8 and 6.4.5-13 : Arrays of the same size are compatible.

Test 6.4.5-9 and 6.4.6-7 : Records of the same size are compatible.

Test 6.4.5-10 : All pointers are compatible with other pointers or the type "Hex".

Test 6.6.2-5 : Any type with a size of less than 128 bytes can be used as a function return type.

Test 6.6.6.3-2 : Trunc and round can have integer parameters.

Test 6.7.2.3-2 : Logical operators are valid for character and integer expressions.

Test 6.7.2.5-6 : Arrays of the same size can be compared. Records of the same size can be compared.

Test 6.8.2.4-2 : Goto between branches of an If statement are allowed.

Test 6.8.2.4-3 : Goto between branches of a Case statement are allowed.

Test 6.8.3.5-7 and 6.8.3.5-8 : Subrange Case statement constants are allowed.

Tests 6.8.3.9-5, 6.8.3.9-6, 6.8.3.9-7, 6.8.3.9-10, 6.8.3.9-12, 6.8.3.9-13, 6.8.3.9-14, 6.8.3.9-15, 6.8.3.9-16, 6.8.3.9-17 : No restrictions are placed on For statement control variable.

Test 6.8.3.9-8 and 6.8.3.9-9 : If a For statement is entered and exited normally the control variable will be valid and contain the final value. If a For statement is not entered then the control variable will be valid and contain the initial value.

Details of deviations

Test 6.1.8-5 : A number can be terminated by a letter.

Test 6.2.1-5 and 6.2.1-6 : Multiple siting for labels is not checked, nor are labels required to be sited at all.

Tests 6.2.2-8, 6.3-6, 6.4.1-3 : Error in scope rules.

Test 6.6.1-7 : Unresolved forward function or procedure declaration is not detected.

Test 6.6.3.3-4 : Use of a field selector as a parameter is not detected.

Test 6.10-4 : No check is made for duplication of program parameters.

Error-handling

Number of errors correctly detected = 19
Number of errors not detected = 31 (13 reasons)

Details of errors not detected

Tests 6.2.1-11, 6.4.3.3-11, 6.4.3.3-12, 6.4.3.3-11, 6.5.4-2, 6.6.2-9 : No checking is made to verify whether or not a variable is accessed that has an undefined value, Instead the variables are guaranteed to contain garbage unless initialized.

Tests 6.4.3.3-1, 6.6.5.3-8, 6.6.5.3-9, 6.6.5.3-10 : Any tagfields or selector variables in a record variant are irrelevant to which variants can be accessed.

Test 6.4.6-10 : No subrange checking on parameter passing.

Tests 6.4.6-12, 6.4.6-13, 6.7.2.4-4 : Overflow checking is done on sets based on byte count - not per element.

Tests 6.5.4-1, 6.6.5.3-4, 6.6.5.3-5, 6.6.5.3-11 : Pointer value is not checked before use.

Tests 6.5.5-2, 6.5.5-3, 6.6.5.3-6, 6.6.5.3-7 : There are no restrictions on the use of pointers or file buffer variables which are currently parameters or elements of a with statement.

Test 6.6.5.2-5 : To support random files a "get" is not executed until called as a procedure or when accessing the file buffer without a valid element - not at the time of "reset".

Test 6.6.6.4-7 : Char and Hex variables "roll over" from maximum value to zero - it is not considered an error.

Test 6.6.6.5-7 : If eof is true - so is eoln - it is not considered an error to check eoln if eof is true.

Tests 6.8.3.5-10 and 6.8.3.5-11 : If no match in case statement, falls through with no error.

Test 6.8.3.9-18 : No restrictions on the control variable of a For loop.

Test 6.8.3.9-1 : At the completion of a For loop the control variable is valid and has the final value.

Tests 6.9.3.2-5 and 6.9.3.2-5 : Writing of real numbers with no digits past the decimal point is permissible.

Quality Measurement

Number of tests run = 52
Number of tests incorrectly handled = 5

Results of tests

"Synthetic Benchmark" - execution time 1 minute, 10 seconds.

"GAMM measure" - execution time 1 minute, 40 seconds for N = 1000

procedure calls - execution time 40 seconds

identifiers are significant up to 120 characters.

source lines may be up to 120 characters.

no reasonable limit on number of real literals allowed.

no reasonable limit on number of strings allowed.

if a line of code is incorrectly part of an unclosed comment the compiler will signal that no code was generated for the line.

at least 50 types may be declared in a program.

no reasonable limit on number of labels, but there can be a maximum of 8 forward referenced goto's in a block.

at least 128 constant definitions are allowed per constant declaration part.

at least 128 procedures are permitted in a program.

maximum size for an array or record or for any variable section is 32750 bytes.

at least 8 index types can appear in an array type.

at least 128 case-constant values are permitted in a variant record.

at least 50 record-sections can appear in the fixed part of a record.

at least 30 distinct variants are permitted in a record.

"Warshall's algorithm" procedure size = 270 bytes, execution time = 9.7 seconds.

considerably less than 300 identifiers are allowed in a declaration list (actual number depends on length of identifier).

at least an 8 dimensional array is allowed.

procedures may be nested to at least 15 levels.

at least 30 formal parameter sections can appear in one parameter list.

the dispose in the standard heap manager will only set the pointer to nil, it will not restore memory.

deeply nested function calls are allowed (at least 6).

deeply nested compound statements are allowed (at least 25).

a procedure may have at least 300 statements.

deeply nested if statements are allowed (at least 25).

at least 256 case constants are allowed.

at least 300 constants are allowed in a case-constant list.

case statements can be nested to at least 15 deep.

repeat loops can be nested to at least 15 deep.

while loops can be nested to at least 15 deep.

for loops can be nested to at least 15 deep.

with statements can be nested to at least 15 deep.

recursive I/O can be used with the same file for the second I/O action.

at least 30 variable-accesses can appear in a read or readln parameter list.

at least 30 write-parameters can appear in a write or writeln parameter list.

data written on the output file appears regardless of the omission of a line marker.

Implementation-defined

Number of tests run = 12
Number of tests incorrectly handled = 1

Details of implementation-defined features

Tests 6.1.9-5 and 6.1.9-6 : alternate symbols are available for comments, array indices, and pointers.

Test 6.4.2.2-10 : Maxint is 32767

Test 6.4.3.4-5 : maximum range of set elements is 0..1007

Test 6.6.6.2-11 : Base = 2, Bits of mantissa = 24, not rounding, minimum value = 2.710506E-20, maximum value = 9.223372E+18

Tests 6.7.2.3-3 and 6.7.2.3-4 : Boolean expressions are fully evaluated.

Test 6.8.2.2-1 and 6.8.2.2-2 : In an assignment statement evaluation of the expression is done before the selection of the variable.

Test 6.8.2.3-2 : When a procedure is called the parameters are evaluated in forward order.

Test 6.9.3.2-6 : Default field widths are : Integers = 10, Boolean = 6, Real = 16, Longinteger = 16, Hex = 6, Longhex = 10.

Test 6.9.3.5.1-2 : Real values written in floating point format have 2 exponent digits.

Test 6.9.3.6-1 : Boolean values written in the default fieldwidth have the format as shown (between quotes) " TRUE" and " FALSE".

Details of tests incorrectly handled

Test 6.6.6.1-1 : functions are not allowed to be passed as parameters to a procedure.

Level 1 Tests - Not applicable

Extensions

Extensions present = 1

Result of extension

Test 6.8.3.5-16 : An otherwise clause is allowed on a case statement.

APPENDIX F - CONVERTING FROM OLDER VERSIONS

To provide the best possible product it is sometimes necessary to abandon previously used methods in favor of new methods. This sometimes results in the need to modify one's code for compatibility with the new methods. As an aid in making this transition, this appendix will describe the changes that are required if the code was written under an older version of the compiler.

Changes from 2.0 to 2.1

1) If using the break function then the compiler comment option {$b+} must be inserted before the standard input device is declared in the program parameters. If the b option is not enabled then a break on the input device will use the normal operating system "crash" mode. It is good practice to turn off the b option after declaring the standard input device if you have turned it on.

2) The S compiler comment option must be enabled if code is to be generated for verifying subrange checking, array indexing, string indexing, and assignments and reads to subrange variables. The R option will now only affect the condition of the runtime error masking flag.

3) The Deverr function returns a byte instead of an integer.

4) The @ symbol was used for absolute addresses, it has now been replaced by the standard identifier "at". The @ is now a equivalent symbol to the ^.

5) The eof function will return true if the device is opened for output only.

6) The eoln function will return true if the device is opened for output only.

7) Variables declared as "external" in 2.0 programs must be declared as "pcr" in 2.1 programs. This does not affect procedure declarations.

8) The memavail function now returns a hex value rather than an integer.

9) Numbers entered without a decimal point or exponent between 32768 and 2147483647 will be longintegers, not real.

10) In the reset procedure, if a string expression is not supplied then a default file name will be used. If the device being reset has its name in the program parameters then the command line argument corresponding with its placement in the program parameter list is used as the file name. If the variable's name is not declared in the program parameters then a null string will be used. In any case if the device is already open then it will be closed before being reset.

11) The rewrite procedure is changed like the reset procedure.

12) In the runtime environment the global and local stack frames have an additional 2 byte entry 10 bytes below the stack mark. This affects the stack setup code (.ps file).

13) In the with statement record variables are searched in the reverse order that they are listed in the with statement. In the case of nested with statements, the innermost with is searched first.

14) In the write and writeln procedures the default fieldwidth for boolean is 6. If the fieldwidth for boolean, hex, or string value is not sufficient to hold its representation then the value will be truncated on the right (the field will not be expanded).

Changes from 2.1 to 2.2

1) The type conversion functions : enchar, floorl, hexint, roundl, sex, str, truncl, valb, valh, vali, vall, and valr, have been removed. They have been replaced by a consistent method of type conversion that uses the desired type as the function name. The following function calls are equivalent to those available in 2.1 :

a) enchar (character)	->	enum (character)
b) enchar (enumerated)	->	char (enumerated)
c) floor (longinteger)	->	integer (longinteger)
d) floorl (real)	->	longinteger (real,floor)
e) hexint (integer)	->	hex (integer)
f) hexint (hex)	->	integer (hex)
g) round (longinteger)	->	integer (longinteger)
h) roundl (real)	->	longinteger (real) or longinteger (real,round)
i) sex (byte)	->	integer (byte)
j) str (type)	->	string (type)
k) trunc (longinteger)	->	integer (longinteger)
l) truncl (real)	->	longinteger (real,trunc)
m) valb (string)	->	boolean (string)
n) valh (string)	->	hex (string)
o) vali (string)	->	integer (string)
p) vall (string)	->	longinteger (string)
q) valr (string)	->	real (string)

EXAMPLES

```
Program printer ;

{ System purpose :
    This system is used to connect three computers to one printer
    on an allocation basis. All three computers and the printer
    use a Centronics interface. After restart of the system no
    computers are allocated to use the printer. To get allocated
    a control code is sent from the computer to the system. This
    control code consists of two fields ; function and time. The
    time field is the lower four bits of the code and indicates
    number of minutes that may elapse without printing from the
    allocated computer before the computer is automatically de-
    allocated. The function field is the upper four bits and is
    as follows :

          $8 : request allocation at 6 LPI line spacing.
          $9 : request allocation at 8 LPI line spacing.
          $A : de-allocate me immediatly.

    If the computer wants de-allocation then the time field must
    be zero. If the computer requests allocation then the time
    field will be the number of minutes unless it is zero, in
    which case the system default time will be used. Even if the
    computer is not allocated the system will acknowledge his
    requests (within one second) but the select line to the
    computer will not be activated until that computer is
    actually allocated to the printer. If the printer is not
    selected then none of the computers will be selected and the
    system will not acknowledge their data transfers.

  System configuration :
    1) ROM : qty. 2 4K * 8 chips at $E000 and $F000 (one used)
    2) RAM : qty. 2 2K * 8 chips at $D000 and $D800
    3) I/O : qty. 2 PIA (6821) at $A00C and $A014

  Printer PIA configuration ($A014) :
    1) PB0-PB7 are data lines to printer
    2) PA0 is select input from printer (active high)
    3) PA1 is form feed pushbutton (active low)
    4) CB1 is ACK input, interupt bit set high on active low
       transistion
    5) CB2 is DS output, pulsed low for 1us after write to PB0-
       PB7
```

```
  Computer PIA configuration ($A00C) :
    1) PB0-PB7 are data lines from enabled computer
    2) PA0-PA1 are computer enable lines :
                          PA0 = 0, PA1 = 0 = computer one
                          PA0 = 1, PA1 = 0 = computer two
                          PA0 = 0, PA1 = 1 = computer three
    3) PA2 is used to reset the enabled computer's select latch -
       indicating that it is not allocated (pulse active low)
    4) PA3 is used to set the enabled computer's select latch -
       indicated that is is allocated (pulse active low)
    5) PA4 is used to ACK the enabled computer (pulse active low)
    6) PA5 is used to reset the enabled computer's DS latch
       (pulse active low)
    7) PA6 is used to read the enabled computers DS latch
       (active high)
    8) CA1 is the real time clock input at 10hz                  }

{********************** CONSTANTS **********************}

  Const
    Print_sel = #1          ; { select from printer on PA0 }
    Print_ff  = #2          ; { form feed button on PA1 }
    Print_ack = #$80        ; { printer ACK on CB1 }
    Print_dira = #0         ; { data direction, side A }
    Print_dirb = #$FF       ; { data direction, side B }
    Print_cntrla = #4       ; { control code, side A }
    Print_cntrlb = #$2C     ; { control code, side B }
    Comp1_enable = #0       ; { enable computer one }
    Comp2_enable = #1       ; { enable computer two }
    Comp3_enable = #2       ; { enable computer three }
    Comp_deselect = #4      ; { deselect computer }
    Comp_select = #8        ; { select computer }
    Comp_ack = #$10         ; { ACK computer }
    Comp_ds_reset = #$20    ; { reset computers DS latch }
    Comp_ds = #$40          ; { read computers DS latch }
    Comp_dira = #$3F        ; { data direction, side A }
    Comp_dirb = #0          ; { data direction, side B }
    Comp_cntrla = #5        ; { control code, side A }
    Comp_cntrlb = #4        ; { control code, side B }
    Comp_data_a = #$3C      ; { normal state for Comp_aux }
    rtc = #$80              ; { real time clock interupt bit }
    Pia_init = #0           ; { initialization for both Pias }
    Qmax = 2999             ; { max element of queue }
    Max_comp = 3            ; { maximum computer number }
    Null = #0               ; { ASCII null character }
    LF = #$A                ; { ASCII line feed character }
    CR = #$D                ; { ASCII carriage return character }
    Form_feed = #$C         ; { ASCII form feed character }
    Cntrl_F = #6            ; { ASCII control F character }
    FF_delay = #5           ; { half a second debounce }
    Sample_time = #10       ; {non allocated computer sample time}
    Minute = 600            ; { number of interupts per minute }
    Sys_default_time = 2    ; {number of minutes if not specified}
    Active = #1             ; { comp_stat indicating active comp. }
```

```
    Idle = #0                ; {comp_stat indicating idle computer}
    Max_ASCII = #$7F         ; { highest valid ascii character }
    Release = #$A0           ; { computer release code }
    Request = #$80           ; { computer request code base }
    Request_mask = #$E0      ; { mask to detect any request code }
    Eight_mask = #$10        ; { mask to check for 8 LPI command }
    Timemask = #$F           ; { mask to leave time from command }
    Enable_mask = #$FC       ; {mask to remove old computer enable}

{********************** VARIABLES ************************}

  Var
    Print_aux : char at $A014          ; { Pia data - side a }
    Print_controla : byte at $A015     ; { Pia control - side a }
    Print_data : char at $A016         ; { Pia data - side b }
    Print_controlb : byte at $A017     ; { Pia control - side b }

    Comp_aux : char at $A00C           ; { Pia data - side a }
    Comp_controla : byte at $A00D      ; { Pia control - side a }
    Comp_data : char at $A00E          ; { Pia data - side b }
    Comp_controlb : byte at $A00F      ; { Pia control - side b }

    Empty : boolean                    ; { Queue empty }
    Full : boolean                     ; { Queue full }
    Select : boolean                   ; { Printer selected }
    Allocated : boolean                ; {A computer is allocated}
    EightLPI : boolean                 ; {Space at 8 lines/inch}
    Comp_aux_set : char                ; { default for Comp_aux }
    Dummy : char                       ; { Temp storage for data }
    Control : char                     ; { Last control code }
    FF : byte                          ; {form feed buttom timer}
    Sample_delay : byte                ; { timer for un-allocated
                                           computers }
    Qin : integer                      ; {put  next character}
    Qout : integer                     ; {get next character}
    Timeout : integer                  ; {no print time out timer}
    Timelimit : integer                ; { Timeout time allowed }
    Current_comp : integer             ; { computer allocated now }
    Comp_stat : Array [0..Max_comp]
                          of byte      ; {status for each computer}
    Q : array [0..Qmax] of char        ; { character queue }

{********************** INITIALIZATION *********************}

  Procedure Init_IO ;
    Begin
      Print_controla := Pia_init ;
      Print_controlb := Pia_init ;
      Print_aux := Print_dira ;
      Print_data := Print_dirb ;
      Print_controla := Print_cntrla ;
      Print_controlb := Print_cntrlb ;
```

```
      Comp_controla := Pia_init ;
      Comp_controlb := Pia_init ;
      Comp_aux := Comp_dira ;
      Comp_data := Comp_dirb ;
      Comp_controla := Comp_cntrla ;
      Comp_controlb := Comp_cntrlb
    End ; { Init_IO }

  Procedure Init ;
    Var
      Comp_init : byte ;
    Begin
      Init_IO ;
      Select := false ;
      Empty := true ;
      Full := false ;
      Allocated := false ;
      Control := null ;
      EightLPI := false ;
      FF := #0 ;
      Qin := 0 ;
      Qout := 0 ;
      Timeout := 0 ;
      For Current_comp := 0 to Max_comp Do
        Comp_stat [Current_comp] := Idle ;
      Sample_delay := #0 ;
      Comp_init := Comp_data_a and not
                   (Comp_ds_reset or Comp_deselect) ;
      Comp_aux := Comp_init or Comp3_enable ;
      Comp_aux := Comp_init or Comp2_enable ;
      Comp_aux := Comp_init or Comp1_enable ;
      Comp_aux := Comp_data_a
    End ;

{********************* QUEUE HANDLERS ********************}

  Procedure Insert (new_char : char) ;
    Begin
      Q[Qin] := new_char ;
      Qin := succ (Qin) ;
      If Qin > Qmax
        then
          Qin := 0 ;
      Full := Qin = Qout ;
      Empty := false ;
      Timeout := 0
    End ;
```

```
  Function Remove : char ;
    Begin
      Remove := Q[Qout] ;
      Qout := succ (Qout) ;
      If Qout > Qmax
        then
          Qout := 0 ;
      Empty := Qin = Qout ;
      Full := false
    End ;

  Procedure Force_8LPI ;
    Begin
      Qout := pred (Qout) ;
      If Qout < 0
        then
          Qout := Qmax ;
      Q[Qout] := Cntrl_F ;
      Full := Qin = Qout ;
      Empty := false
    End ;

  Procedure Put_string (line : string [40]) ;
    Var
      ccnt : integer ;
    Begin
      For ccnt := 1 to length (line) Do
        Insert (line[ccnt])
    End ;

{************* REAL TIME CLOCK INTERUPT HANDLER ************}

  Procedure rtcirq ; entry ;
    Var
      Comp_clear : byte ;
    Begin
      Comp_clear := Comp_aux ; { clear interupt }
      If Select
        then
          begin
            If FF <> #0
              then
                FF := FF - #1 ;
            If Sample_delay <> #0
              then
                Sample_delay := Sample_delay - #1 ;
            If Allocated and empty
              then
                Timeout := Timeout + 1
          end
    end ; { RTCIRQ }

  Procedure setirq ; external ;
  Procedure irqoff ; external ;
```

```
{********************* COMPUTER PORT SUPPORT ******************}

  Procedure Select_mode (Mode : byte) ;
    Begin
      If Allocated
        then
          begin
            Comp_aux_set := Comp_data_a +
                            chr (Current_comp -1) ;
            Comp_aux := Comp_aux_set and not Mode ;
            Comp_aux := Comp_aux_set
          end
    End ; { Select_mode }

  Procedure Allocate (Comp_num : integer ; C_code : byte) ;
    Begin
      Current_comp := Comp_num ;
      Allocated := true ;
      Comp_stat [Current_comp] := Active ;
      Select_mode (Comp_select) ;
      EightLPI := C_code and Eight_mask <> #0 ;
      C_code := C_code and timemask ;
      If C_code = #0
        then
          Timelimit := Sys_default_time * Minute
        else
          Timelimit := ord (C_code) * Minute ;
      Timeout := 0
    End ; { Allocate }

  Function Fetch (Default : byte) : char ;
    Begin
      irqoff ;
      Fetch := Comp_data ;
      Comp_aux := Default and not
                  (Comp_ds_reset or Comp_ack) ;
      Comp_aux := Default ;
      setirq
    End ;
```

```
{****************** ALLOCATED COMPUTER HANDLER ****************}

  Procedure Check_allocated ;
    Var
      EOJ : String [30] ;
      ccnt : integer ;
    Begin
      If Timeout > Timelimit
        then
          Control := Release ;
      If Control <> null
        then
          begin
            If Control = Release
              then
                begin
                  EightLPI := false ;
                  For ccnt := 1 to 40 Do
                    begin
                      Insert (CR) ;
                      Insert (LF)
                    end ;
                  Put_string ('******************************') ;
                  Put_string (' END OF JOB - COMPUTER #') ;
                  Insert ('0' + chr(Current_comp)) ;
                  Put_string (' ******************************') ;
                  Insert (CR) ;
                  Insert (LF) ;
                  Insert (Form_feed) ;
                  Select_mode (Comp_deselect) ;
                  Allocated := false ;
                  Comp_stat [Current_comp] := Idle
                end ;
            If Control and Request_mask = Request
              then
                Allocate (Current_comp, Control) ;
            Control := null
          end
    End ; { Check_allocated }
```

```
{*************** UNALLOCATED COMPUTER HANDLER ***************}

  Procedure Check_unallocated ;
    Var
      Comp_scan : byte ;
      Idle_scan : integer ;
    Begin
      Sample_delay := Sample_time ;
      For Idle_scan := 1 to Max_comp Do
        If Comp_stat [Idle_scan] <> Active
          then
            begin
              Comp_scan := Comp_aux and enable_mask or
                                chr (Idle_scan - 1) ;
              Comp_aux := Comp_scan ;
              If Comp_aux and Comp_ds <> #0
                then
                  begin
                    Control := Fetch (Comp_scan) ;
                    If Control = Release
                      then
                        Comp_stat [Idle_scan] := Idle
                      else
                        If Control and Request_mask
                                  = Request
                          then
                            Comp_stat [Idle_scan] := Control
                  end ;
              Control := Comp_stat [Idle_scan] ;
              If (Control and Request_mask = Request)
                            and not Allocated
                then
                  Allocate (Idle_scan, Control)
            end ;
      Comp_aux := Comp_aux_set ;
      Control := null
    End ; { Check_unallocated }
```

```
{************************ MAIN PROGRAM ************************}

Begin { Printer }
  Init ;
  setirq ;
  Comp_aux_set := Comp_data_a ;
  Repeat
    If Select
      then
        If Print_aux and Print_sel <> #0
          then
            begin
              If not Full and (Print_aux and Print_ff = #0)
                then
                  Begin
                    If FF = #0
                      then
                        Insert (Form_feed) ;
                    FF := FF_delay
                  End ;
              While not Empty and (Print_controlb and Print_ack
                                   <> #0) do
                begin
                  Dummy := Print_data ; { clear interupt }
                  Dummy := Remove ;
                  Print_data := Dummy ;
                  If (Dummy = LF) and EightLPI
                    then
                      Force_8LPI
                end ;
              If Allocated
                then
                  begin
                    While (Comp_aux and Comp_ds <> #0)
                                and not Full Do
                      begin
                        Dummy := Fetch (Comp_aux_set) ;
                        If Dummy <= Max_ASCII
                          then
                            Insert (Dummy)
                          else
                            Control := Dummy
                      end ;
                    Check_allocated
                  end ;
              If Sample_delay = #0
                then
                  Check_unallocated
            end
```

```
          else
            begin
              Select := false ;
              Select_mode (Comp_deselect)
            end
      else
        If Print_aux and Print_sel <> #0
          then
            begin
              Select := true ;
              Select_mode (Comp_select) ;
              Print_data := Null
            end
  Until false
End. { Printer }
```

The next section is the stack setup code, modified from what was output from the linkage creator.

```
        NAM START
        XDEF START
        XREF PRINTE
START   LDS #$D100 SET SYSTEM STACK
        LDU #$DFE6 SET DATA STACK
        LEAY 10,U SET GLOBAL STACK MARK
        LDX #$D100
        STX -2,Y SET DATA STACK LIMIT
        STX 2,Y SET HEAP POINTER
        LDX #$D000
        STX -4,Y SET SYSTEM STACK LIMIT
        STY -6,Y SET GLOBAL MARK POINTER
        CLR -7,Y DISABLE RUNTIME ERROR CHECKING
        LBRA PRINTE
        END
```

The next section handles the two assembly language procedures for turning on and off the interupt mask and the code for handling the interupt.

```
        NAM PSUP
        TTL PRINTER SUPPORT
        XDEF SETIRQ,IRQVEC,IRQOFF
        XREF RTCIRQ
SETIRQ ANDCC #$EF
        RTS
IRQOFF ORCC #$10
        RTS
IRQVEC LDY #$DFF0 SET GLOBAL STACK MARK
        LDA #-1 CALLING LEX2 FROM LEX1
        LBSR RTCIRQ CALL PASCAL PROCEDURE
        RTI
        END
```

The last part specifies the interupt and restart vectors

```
        NAM INTRPT
        TTL INTERUPT VECTOR FOR PRINTER
*
* THIS CODE MUST RESIDE IN ROM STARTING AT $FFF8
*
        XREF START,IRQVEC
        XDEF RESET,IRQ
IRQ     FDB IRQVEC
        FDB 0,0
RESET   FDB START
        END
```

The runtime error printing device was changed to :

```
.ERROR JMP [$FFFE]
```

which of course will restart the program. Using the OmegaSoft linker the commands looked like :

```
STRP=$F000
LOAD=PRINTER.PA PRINTER.CA PSUP
LIB=RL
CURP=$FF8
LOAD=INTRPT
OBJA=PRINTER
MAPSM
EXIT
```

```
program rotate_page (input, output) ;
  type
    line = record
             link : ^line ;
             str : string
           end ;
  var
    start_frame : ^integer ;
    page_st, page_ptr : ^line ;
    i, longest : integer ;
  begin
    repeat
      mark (start_frame) ;
      new (page_st) ;
      page_ptr := page_st ;
      writeln ('Enter lines (return to rotate, / to exit)') ;
      longest := 0 ;
      readln (page_ptr^.str) ;
      if page_ptr^.str = '/'
        then
          exit ;
      while page_ptr^.str <> '' do
        begin
          new (page_ptr^.link) ;
          page_ptr := page_ptr^.link ;
          readln (page_ptr^.str) ;
          if length (page_ptr^.str) > longest
            then
              longest := length (page_ptr^.str)
        end ;
      page_ptr^.link := nil ;
      for i := 1 to longest do
        begin
          page_ptr := page_st ;
          while page_ptr <> nil do
            begin
              if i > length (page_ptr^.str)
                then
                  write ('  ')
                else
                  write (page_ptr^.str [i],' ') ;
              page_ptr := page_ptr^.link
            end ;
          writeln
        end ;
      release (start_frame)
    until false
  end.
```

```
program runtime (output);

  {
    this program reads a merged runtime library file and sorts
    the file by subroutine name. then it prints the contents with
    the corresponding module/file name. This program is designed
    to run under OS-9, however with a few changes it will also
    run under DOS69 and FLEX. This program was donated by K.E. of
    Smoke Signal Broadcasting, makers of really fine stuff.
  }

  type
    struc = record
      sub_name : string[6];
      mod_name : string[6]
    end;

    sptr = ^struc;

  var
    count : byte;
    loop : integer;
    first, current, last : ^struc;
    table_start, table_end, ptr : ^sptr;
    name : string[6];
    line_buf : string[126];
    f : file of byte;

  procedure build_list;
    begin
      line_buf := substr (line_buf, 2, 126);
      while (line_buf[0] > #0) do
        begin
          case line_buf[1] of
            #0 : line_buf := substr (line_buf, 18, 126);
            #$30 : line_buf := substr (line_buf, 8, 126);
            #$24 :
              begin
                new (last);
                with last^ do
                  begin
                    sub_name := substr (line_buf, 2, 6);
                    mod_name := name;
                  end;
                line_buf := substr (line_buf, 10, 126)
              end
        end
      end
    end;
```

```
procedure sort;

  procedure exchange (ptr1, ptr2 : ^sptr);
    var
      temp : ^sptr;
    begin
      temp := ptr1^;
      ptr1^ := ptr2^;
      ptr2^ := temp
    end;

  procedure quick (lo, hi : ^sptr);
    var
      i, j, pivit : ^sptr;
    begin
      if lo < hi
        then
          begin
            i := lo;
            j := hi;
            pivit := j;
            repeat
              while (i < j)
                    and (i^^.sub_name <= pivit^^.sub_name) do
                i := i + $2;
              while (j > i)
                    and (j^^.sub_name >= pivit^^.sub_name) do
                j := j - $2;
              if i < j
                then
                  exchange (i, j);
            until i >= j;
            exchange (i, hi);
            if i - lo < hi - i
              then
                begin
                  quick (lo, i - $2);
                  quick (i + $2, hi)
                end
              else
                begin
                  quick (i + $2, hi);
                  quick (lo, i - $2)
                end
          end
    end;
```

```
    begin
      mark (table_start);
      for ptr := $0 to (last - first)
                      div hex(sizeof(struc)) do
        begin
          new (table_end);
          table_end^ := ptr * hex(sizeof(struc)) + first
        end;
      quick (table_start, table_end)
  end;

begin
  mark (first);
  last := nil;
  reset (f, cline(1));
  while not eof(f) do
    begin
      read (f, count);
      read (f, count);
      for loop := 1 to ord (count + #1) do
        read (f, line_buf[loop]);
      line_buf[0] := chr (loop - 2);
      case line_buf[1] of
        '2' : name := substr (line_buf, 3, 6);
        '3' : build_list
      end
    end;
  close (f);
  if last <> nil
    then
      begin
        sort;
        writeln; writeln;
        ptr := table_start;
        loop := 0;
        repeat
          current := ptr^;
          with current^ do
            begin
              write (sub_name, ' : ', mod_name, '  ');
              loop := succ (loop);
              if loop = 4
                then
                  begin
                    writeln;
                    loop := 0
                  end
            end;
          ptr := ptr + $2
        until ptr > table_end;
        page
      end
end.
```

```
program avail (input, output) ;

{
  This example demonstrates the effect of subroutine calls
  on the size of the free heap space
}
  var
    max_level : integer ;

  procedure recurse (call_level, max_level : integer) ;
    begin
      writeln ('Entering procedure level ', call_level:0,
               ', heap space available = $', memavail:4) ;
      if call_level < max_level
        then
          recurse (call_level + 1, max_level) ;
      writeln ('Exiting procedure level ', call_level:0,
               ', heap space available = $', memavail:4)
    end ;

  begin
    write ('Enter maximum procedure nesting level : ') ;
    read (max_level) ;
    if max_level > 0
      then
        recurse (1, max_level)
  end.


{
  function to convert a 3 byte array into a longinteger (used to
  convert OS-9 byte counts into longinteger for seek parameter).
}
  function convert (arry : array [0..2] of byte) : longinteger ;
    type
      trick = record
                case boolean of
                  false : (ary : record
                                   i : byte ;
                                   j : array [0..2] of byte
                                 end) ;
                  true : (long : longinteger)
              end ;
    var
      trec : trick ;
    begin
      trec.ary.i := #0 ;
      trec.ary.j := arry ;
      convert := trec.long
    end ;
```

```
{
  Example of duplicating a text path (such as the standard
  output path) except making it a file of char instead.
}
  var
    dup : file of char ;
          .
          .
  procedure set_path ;
    type
      descriptor = record
                     mode : byte ;
                     err : byte ;
                     drv : hex ;
                     elnt : integer ;
                     elmt : char ;
                     path : byte {for OS-9 only}
                   end ;
    var
      main, copy : ^descriptor ;
    begin
      main := addr (output) ;
      copy := addr (dup) ;
      copy^ := main^ ;
      copy^.mode := main^.mode and #%11111011 {turn of M$TEXT}
    end ;
```

DEBUGGER

DEBUGGER OPERATION

Command line syntax :

DB <program file name> {<module file name>} [D=<hex>] [H=<hex>]

The D option will override the default debugger symbol table size. The H option will override the default Heap start address (see chapter 15 for operating system specifics).

The debugger operates in two phases to maximize the amount of memory available for your program. At the start of phase one the available memory area is divided into sections by the kernel, refer to chapter 15 in this manual if you are interested in this segmentation. The debugger table size is defaulted to a reasonable size or may be specified using the D command line option. If the debugger table size changes, the code area will change by the reverse amount. Making the debugger table smaller will allow more room to be shared by your program and its heap/data stack.

The first file specified in the command line must be a Pascal program compiled using the compiler 'D' command line option. Any additional files specified must be Pascal modules compiled using the compiler 'D' or 'I' command line option. The files specified on the command line are assembled into the code area in one pass. A large disk buffer is used to speed this operation.

After assembly is completed the module linker will be executed. If there was more than one file specified on the command line the linker will resolve any references (external and entry procedures or variables) between modules. The number of still unresolved references is then displayed and a menu of options is supplied. If you are not using any assembly language support for your program (other than the standard runtime routines) then there will be no unresolved references and you can simply type a carriage return, skip the following discussions and proceed to the "Debugger Commands" section of this manual.

In response to the menu prompt the following responses are accepted (all terminated by a carriage return) :

1) display unresolved symbols - names of symbol references that have no matching definitions are displayed. These would include variables declared as external that do not have a matching variable declared as entry in another module, variables declared as pcr, or procedures declared as external that do not have a matching procedure declared as entry in another module.

2) display resolved symbols - names, addresses, and relocation type. In the case of external variables the relocation type will be 'A' for absolute and the address listed will be its stack offset. In the case of external procedures or pcr variables the the relocation type will be 'P' for program relative and the address listed will be its actual address.

3) set display path - allows entry of a device or file name to be the destination of future display commands. This path remains in effect until either this phase is exited or by using this command to change the path.

4) read symbol file - allows entry of a file name that contains the names and addresses of symbol definitions. This feature is useful when you must make changes in a Pascal program and debug frequently using the same already loaded assembly language routines. The file must be a standard text file with lines in the following format :
NAME=HEX ADDRESS

5) read symbols from terminal - allows entry of symbol definitions from the terminal. Entry is terminated by entering a line that contains only a carriage return. The format is one line at a time in the form :
NAME=HEX ADDRESS

CR) exit this phase - loads the debugger overlay and starts phase two (actual debugging). If you enter this command while there are still unresolved symbols then it will not be allowed to proceed to phase two, you can either go back and resolve those symbols or exit the debugger.

Refer to the chapter 15 for more information regarding use of assembly language routines with the debugger on your operating system.

DEBUGGER COMMANDS

BREAKPOINT

Format: B [line #]

This command will set a breakpoint or display the current breakpoints. If no number (or zero) is entered, then the current breakpoints are displayed. If a non-zero number is entered then a breakpoint is set at the Pascal line if it is an executable line. Executable lines are marked with an "*" (instead of ":") after the line number on the Pascal compilation listing. If a breakpoint is hit while running the program, the message :

Breakpoint encountered at line <line#>

is displayed and the debugger is entered. The breakpoint is not removed from the table. The maximum number of breakpoints that can be set at any one time is 16.

CHANGE VARIABLE

Format: Cx <expression>

The change command is used to change a variable. Refer to the 'Display Variable' section for the allowable values of 'x' and the format of the expression. The new value for the variable is entered on the next line. For the common variable types the new value should be entered exactly as if were being read by a Pascal program using the "read" statement (because it is). A byte variable is entered as one hex byte. A structured variable is entered as a sequence of hex bytes separated by spaces (only one line may be input). Sets are entered (or modified) using a menu-driven scheme that allows :

1) add elements - element values entered are added to the set.
2) remove elements - element values entered are removed from the set.
3) make empty set - the set is set to no elements present.
4) display current set - displays what you have so far.
CR) done - exits change command.

The set elements may be entered as decimal numbers, hex numbers (preceded by a $), or characters (preceded by a ').

DISPLAY VARIABLE

Format: Dx <expression> [:length]

where x can be :	**blank - use default type of variable**	
	B - boolean	**C - char**
	H - hex	**I - integer**
	L - longinteger	**R - real**
	S - string	**T - set**
	U - byte	**X - structure**
	E - longhex	

<expression> may contain variable names, integer numbers, hex numbers (preceded by $), the symbol "@" which represents the results of the last expression, the operators "+","-","*", and "^" (pointer dereferencing and device element display), and parenthesis. The precedence is :

```
highest:  ( )
            ^
            *
lowest:   + -
```

Operators of the same precedence are evaluated from left to right except "^" which binds from right to left.

If [:length] is specified it must be an integer and only affects how many bytes of a structure are displayed.

Only variables that are valid at the current line number in the program can be displayed (following standard scoping rules). If any operators are used in the expression the variable name should be the first value in the expression - since this is what sets the default display type.

'x' defines the format of the variable to be displayed. Sets are displayed by listing the elements which are present in the set. You have the option of displaying those elements as integers, hex numbers, or characters. The display structure format is similar to a memory dump in many monitors. The start address is displayed followed by a maximum of 16 bytes (in hex) per line, followed by the byte's ASCII equivalent characters (Non-printable characters shown as '.'). This format is normally used to display arrays, records, or devices. Byte variables are displayed in both hex and integer. This form is useful for displaying enumerated types and string or set dynamic lengths. Character and string variables are displayed surrounded by single quotes.

ENTRY FLAG

Format: E "ON" or "OFF" or "+" or "-"

This command turns the Procedure entry print flag on (ON or +) and off (OFF or -). If the flag is on, then when entering a procedure the following message is printed :

Enter Procedure <procedure name>

The enter message is displayed before the first executable line in the procedure. The state of this flag is displayed as part of the Debugger status message (see Status).

GO

Format: G

Go starts execution from the beginning of the program.

HELP

Format: H

This command displays the following helpfulness:

```
Debugger command summary
General control     General display     Variable display
---------------     ---------------     ------------------------------
Go                  Status              Dx variable accessing
Proceed             Help                   expression [:length]
Quit                N <proc name>           x can be :
Trace one           E ON, E OFF,            blank - use default type
Trace N             E+, E- procedure        B - boolean    C - char
                    or function             H - hex        I - integer
Breakpoints         entry display           L - longint    R - real
---------------     L ON, L OFF,            S - string     T - set
B display all       L+, L- line             U - byte       X - structur
B N  set            number display          E - longhex
R remove all        Update Cline        Cx Change variable
R N  remove
```

LINE FLAG

Format: L "ON" or "OFF" or "+" or "-"

This command turns the line number print flag on (ON or +) and off (OFF or -). If the flag is on, then the line number is printed before the line is executed in the following format :

Line #<line number>

The state of this flag is displayed as part of the Debugger status message (see Status)

NUMBER

Format: N <procedure name> or <PROGRAM>

This command displays the line number of the specified procedure or program's first executable line (Begin). This is useful when debugging a program using a non-current listing.

PROCEED

Format: P

Proceed continues execution of the program from the current line. The program must have previously encountered a breakpoint or a trace completion (must not have exited the program normally or encountered a runtime error).

QUIT

Format: Q

Quit causes the debugger to exit.

REMOVE BREAKPOINT

Format: R [line #]

This command removes breakpoints from the breakpoint table. If no number (or zero) is entered, then all breakpoints are removed. If a non-zero number is entered, then only the breakpoint at that line is removed.

STATUS

Format: S

This command displays the status of your program and the debugger. The status of the program is not displayed if the program isn't being executed. The program status contains :

Current line number	<integer>
Lex level	<integer>
Trace count	<integer>
Current location	<hex>
Stack	<hex>
Local stack mark	<hex>
Global stack mark	<hex>
Heap pointer	<hex>

Current line number (and corresponding current location) is where your program last hit a breakpoint or a trace completion. If the debugger was entered by an error in your program then the line number may contain "????" indicating that it is not at the start of a line. Lex level is the current lexical level (Global is lex level 1). Stack is the current value of the User data stack. Local stack mark is the current base of any local variables, whereas Global variables are referenced from the global stack mark. The trace count is zero if there is not a trace in progress, otherwise it is the number of statements to complete the trace.

The debugger status contains :

Procedure entry/exit	"on" or "off"
Line number display	"on" or "off"
Cline ('contents')	'string'
Program start	<hex>
Program end	<hex>
Code area	<integer>%
Debugger table	<integer>%

Procedure entry/exit and Line number display reflect the state of the flags set by the Entry (E) and Line (L) commands. Cline is the current value of the command line (can be modified by the Update Cline command). Code area is the percentage of the allowable memory used by your program. The area available is determined by the total system memory size and the size of the debugger table. Normally the heap pointer is set right past the end of the program (unless set by the heap command line option). This unused area beyond the actual end of the program is added to the space available for the heap and data stack. Debugger table is the percentage of the allowable memory used by the debugger table. The area available is determined by the "D" debugger command line option or defaulted by the debugger.

TRACE

Format: T [Trace count]

This command traces one or more statements from the occurrence of a breakpoint or previous trace completion and then returns to the debugger. If no trace count (or zero) is entered, then one statement is traced, otherwise the specified trace count is used. The current trace count is displayed as part of the debugger status message. When the end of the trace is reached the following message is displayed and the debugger is entered :

Trace finished at line number <Line #>

UPDATE COMMAND LINE

Format: U

This command changes the current command line. The command line is set to the string entered on the next line. The current value of the command line is displayed as part of the debugger status message. This string is accessed when your program executes the Cline function or opens a file using the default file name (file listed in program parameters).